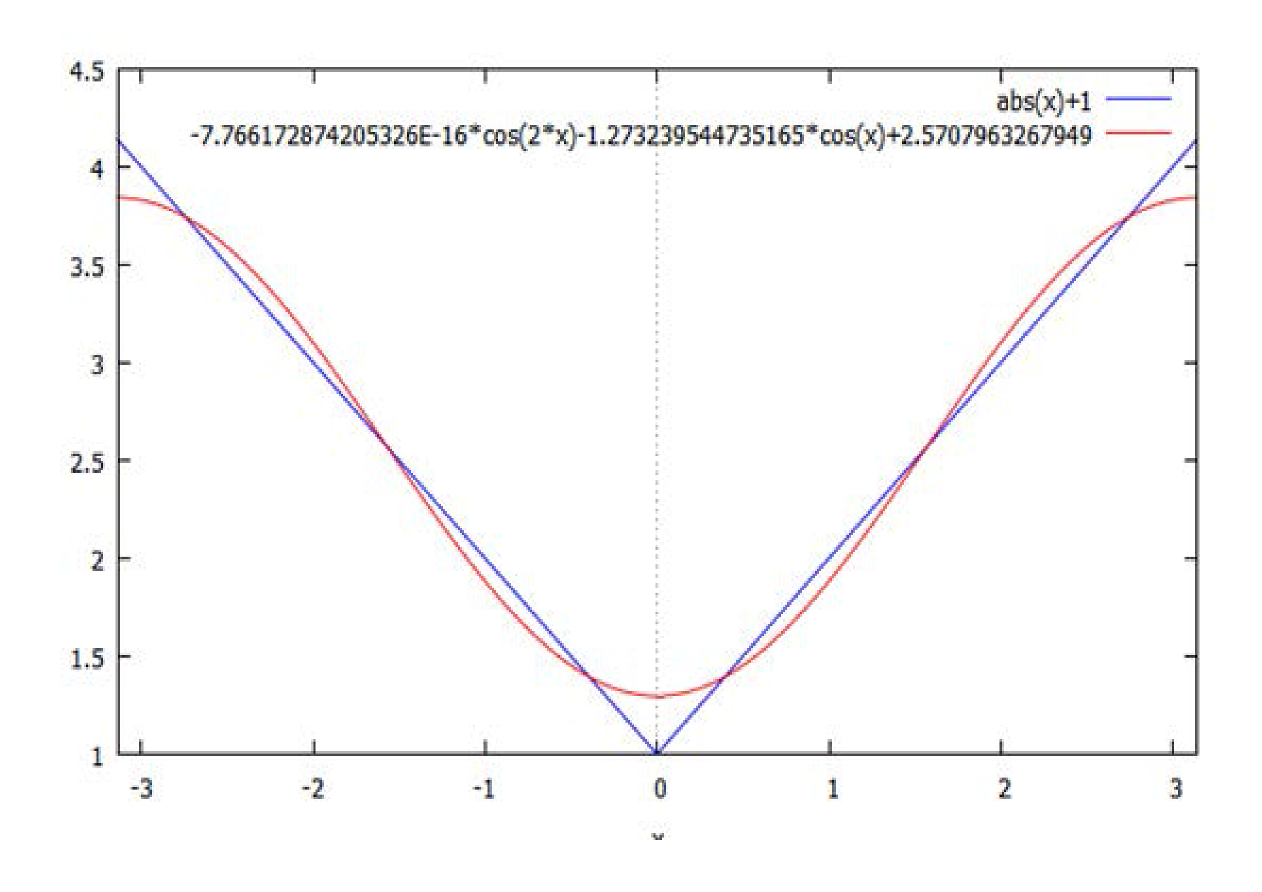


PRÁCTICAS DE CÁLCULO NUMÉRICO CON MAXIMA



Antonio Vigueras Campuzano
Departamento de Matemática Aplicada y Estadística
Escuela Técnica Superior de Ingeniería Industrial
UNIVERSIDAD POLITÉCNICA DE CARTAGENA





PRÁCTICAS DE CÁLCULO NUMÉRICO CON MAXIMA

Antonio Vigueras Campuzano





CRAI Biblioteca
Plaza del Hospital, 1
30202 Cartagena
968325908
ediciones@upct.es

Primera edición, 2016

ISBN: 978-84-608-7868-1

Imagen de la cubierta: Representación gráfica con Maxima



Índice general

Prólogo

La asignatura Cálculo Numérico se plantea como una introdución al estudio de los métodos numéricos. En el caso del Grado en Ingeniería en Tecnologías Industriales de la Universidad Politécnica de Cartagena, se estudia en el primer cuatrimestre del tercer curso. Se trata de una asignatura de 6 ECTS, cuyo objetivo es que el alumno conozca y sepa aplicar los métodos numéricos básicos en situaciones concretas propias de su titulación, así como capacitarle para que pueda preparar y manejar algoritmos y programas de cálculo para la resolución de problemas prácticos, a la vez que comprenda las posibilidades y limitaciones de las técnicas numéricas utilizadas.

Dentro de las actividades presenciales de la asignatura, está prevista la realización de seis sesiones prácticas, de dos horas de duración cada una en el aula de informática, trabajando con el software libre Maxima, bajo la interface gráfica wxMaxima, los estudiantes pueden disponer libremente de este software en http:maxima.souerceforge.net/es/. Para la realización de los programas contenidos en este manual hemos utilizado, concretamente, las versiones 5.28.0-2 de Maxima y 12.04.0 de wxMaxima. En estas sesiones, realizamos algoritmos y programas propios para los métodos fundamentales desarrollados, utilizando dicho software. Con esta finalidad hemos preparado este manual. En estas prácticas se persiguen los tres objetivos fundamentales siguientes:

- Saber realizar, utilizar y depurar programas de algoritmos diseñados en la parte teórica de la asignatura.
- Estudiar el comportamiento numérico de los códigos programados.
- Dotar al alumno de criterios para seleccionar un algoritmo para un problema concreto.

Se proponen las sesiones prácticas, de dos horas de duración cada una, siguientes:

1. Introducción al entorno wxMaxima: primera parte.

2. Introducción al entorno wxMaxima: segunda parte.
3. Resolución de ecuaciones y sistemas no lineales.
4. Métodos directos e iterativos de resolución de sistemas lineales.
5. Interpolación. Derivación e integración numérica.
6. Métodos numéricos de integración de PVI para EDO's.

Cada una de las cuales va seguida de una serie de ejercicios propuestos, que esperamos que el alumno intente resolver por si mismo antes de ver las soluciones que le aportaremos.

En cuanto a la bibliografía de prácticas de Cálculo Numérico con Maxima, que no es muy abundante, he consignado tan sólo la realmente utilizada para preparar este manual y que se cita al final del mismo, espero que su consulta pueda servir al alumno para profundizar más en los temas tratados así como para abordar algunos de los trabajos que se le puedan proponer a lo largo del curso, cabe destacar el manual oficial de Maxima, la ayuda en línea de dicho programa, así como el texto [10], en el que se desarrollan sobradamente todos los contenidos teóricos abordados en estas prácticas y que puede consultarse, si hubiera dudas en alguno de los métodos utilizados en estas prácticas.

Considero que este manual de prácticas puede ser útil para estudiantes de otras titulaciones de ciencias o ingeniería, cuyo plan de estudios contenga alguna asignatura de introducción a los métodos numéricos.

Cartagena, marzo de 2016

Capítulo 1

Introducción a Maxima: Parte I

1.1. ¿Qué es Maxima?. Operaciones aritméticas

Maxima es un sistema para la manipulación de expresiones simbólicas y numéricas, incluyendo diferenciación, integración, desarrollos en series de Taylor, transformadas de Laplace, ecuaciones diferenciales ordinarias, sistemas de ecuaciones lineales, y vectores, matrices y tensores. Maxima produce resultados con alta precisión usando fracciones exactas y representaciones con aritmética de coma flotante arbitraria. También realiza representaciones gráficas de funciones y datos en dos y tres dimensiones.

Es un sistema multiplataforma y su código fuente puede ser compilado sobre varios sistemas incluyendo Windows, Linux y MacOS X. El código fuente para todos los sistemas y los binarios precompilados para Windows y Linux están disponibles en el Administrador de archivos de SourceForge.

Maxima es un descendiente de Macsyma, el legendario sistema de álgebra computacional desarrollado a finales de 1960 en el instituto tecnológico de Massachusetts (MIT). Este es el único sistema basado en el esfuerzo voluntario y con una comunidad de usuarios activa, gracias a la naturaleza del open source. Macsyma fue revolucionario es sus días y muchos sistemas posteriores, tales como Maple y Mathematica, estuvieron inspirados en él.

La rama Maxima de Macsyma fue mantenida por William Schelter desde 1982 hasta su muerte en 2001. En 1998 él obtuvo permiso para liberar el código fuente bajo la licencia pública general(GPL) de GNU. Gracias a su esfuerzo y habilidad, Maxima fue posible y estamos muy agradecidos con él por su dedicación voluntaria y su gran conocimiento por conservar el código original de DOE Macsyma vivo. Desde su paso a un grupo de usuarios y desarrolladores, Maxima ha adquirido una gran audiencia.

Maxima está en constante actualización, corrigiendo bugs y mejorando el código y la documentación. Debido, en buena parte, a las sugerencias y contribuciones de su comunidad de usuarios.

Información acerca de cómo conseguirlo, instalarlo y demás puede encontrarse en http://maxima.sourceforge.net/es/. Se puede trabajar con Maxima en línea de comandos desde la consola, pero hay dos opciones gráficas, una de ellas llamada Maxima Algebra System (xmaxima) es bastante antiestética, mientras que wxMaxima (wxmaxima) es bastante más agradable a la vista y será la opción que utilizaremos en este curso. Centrémonos pues en esta última opción gráfica wxMaxima, que se puede descargar de su página web, http://wxmaxima.sourceforge.net/, aunque suele venir incluido con Maxima. En ella, al igual que en otros CAS como Mathematica, se trabaja a partir de celdas, que pueden contener más de una línea de ejecución. La entrada (en inglés, input) queda asignada a una expresión %i) junto con un número, mientras que la salida (output en inglés) aparece como %o) y el mismo número. Nótese que en wxMaxima la tecla Intro por sí sola cambia de línea, para ejecutar una celda es necesario utilizar la combinación Mayúsculas+Intro (shift+enter). Este comportamiento se puede intercambiar editando las preferencias de wxMaxima.

Cada sentencia o comando de Maxima puede finalizar con punto y coma (;) (wxmaxima escribe automáticamente el último ;) o bien con dólar ($). Se pueden escribir varias sentencias una a continuación de otra, cada una de las cuales finaliza con su correspondiente punto y coma (;) o dólar ($), y pulsar shift+enter al final de las mismas para ejecutarlas. En el caso del punto y coma, aunque aparezcan escritas en la misma línea cada una de tales sentencias aparecerá con una etiqueta diferente. En cambio si se usa el dólar como finalización de sentencia, las operaciones que correspondan serán realizadas por Maxima, pero el resultado de las mismas no será mostrado, salvo la última.

Cuando arrancamos Maxima aparece una página en blanco en la que podemos escribir las operaciones que queremos que realice, una vez escritas hemos de pulsar shift+enter o bien el enter del teclado numérico para ejecutar, entonces aparecerá lo que sigue:

```
(%i1) 2+4;
(%o1) 6
(%i2) 3*5;
(%o2) 15
```

```
(%i3)  7/2;
```

(%o3) $\frac{7}{2}$

```
(%i4)  7.0/2;
```

(%o4) 3,5

```
(%i5)  7.0/2.0;
```

(%o5) 3,5

```
(%i6)  7/3;
```

(%o6) $\frac{7}{3}$

```
(%i7)  7/3.0;
```

(%o7) 2,333333333333334

```
(%i8)  5!;
```

(%o8) 120

```
(%i9)  3^4;
```

(%o9) 81

```
(%i10) 3**4;
```

(%o10) 81

Los %i1, %o1, etc., los escribe el programa para indicar el número de entrada o salida correspondiente. Las operaciones aritméticas que realiza son las siguientes:

suma: $x + y$	resta: $x - y$	producto: $x * y$
división: x/y	factorial: x!	potencia: x^y o bien $x * *y$

Si escribimos, por ejemplo

```
(%i11) 4^200;
```

(%o11) 2582249878086908589655919172 00[61$digits$]28013783143590317197 2747493376

El resultado no pone todos los dígitos, de hecho nos informa que ha omitido 61 dígitos. Para saber los dígitos omitidos vamos al menú Maxima,

Cambiar pantalla 2D y escogemos ascii, si ahora repetimos la operación obtendremos:

```
(%i12) set_display('ascii)$
```

```
(%i13) 4**200;
```

(%o13)25822498780869085896559191720030118743297057928292235128306593565406476220168411946296453532801378314359031719727474933376

```
(%i14) 2^500;
```

(%o14)3273390607896141870013189696827599152216642046043064789483291368096133796404674554883270092325904157150886684127560071009217256545885393053328527589376

```
(%i15) 2.5*2;
```

(%o15)5,0

Maxima es un programa de cálculo simbólico y hace las operaciones encomendadas de forma exacta, por ejemplo la suma de fracciones devuelve otra fracción y lo mismo la raíz cuadrada, a no ser que se le pida usando las sentencias: "**float(número)**" que da la expresión decimal de número con 16 dígitos. La instrucción "**número, numer**" también da la expresión decimal de número con 16 dígitos, en tanto que "**bfloat(numero)**" da la expresión decimal larga de número acabada con b seguido de un número n, que significa multiplicar por 10^n. La precisión que nos brinda el programa se puede modificar con la instrucción "**fpprec: m**" que indica el número de dígitos a utilizar. Veamos algunos ejemplos:

```
(%i16) %pi;
```

(%o16) $\%pi$

```
(%i17) 3^200;
```

(%o17)265613988875874769338781322035779626829233452653394495974574961739092490901302182994384699044001

```
(%i18) 3/7+5/6;
```

(%o18) $\frac{53}{42}$

```
(%i19) float(%pi);
```

($\%o19$)3,141592653589793

```
(%i20) %e;
```

($\%o20$) $\%e$

```
(%i22) float(%e);
```

($\%o22$)2,718281828459045

```
(%i23) float(3^200);
```

($\%o23$)2,6561398887587475 $E + 95$

```
(%i24) float(3/7+5/6);
```

($\%o24$)1,261904761904762

```
(%i25) bfloat(%pi);
```

($\%o25$)3,141592653589793 b0

```
(%i26) fpprec: 100;
```

($\%o26$)100

```
(%i27) bfloat(%pi);
```

($\%o27$)3,14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706 $8b$0

Para volver al modo por defecto de pantalla, vamos al menú Maxima, Cambiar pantalla 2D y escogemos xml, como figura a continuación

```
(%i28) set_display('xml)$
```

```
(%i29) bfloat(%pi);
```

(%o29) 3,14159265358979323846264338 32[43 $digits$]628620899862803482534211706 $8b$0

```
(%i29) bfloat(%pi);
```

(%o29) 3,14159265358979323846264338 32[43 $digits$]628620899862803482534211706 $8b$0

```
(%i30) set_display('ascii)$
```

Asignaciones, el operador “:” se utiliza para asignar a una variable el

valor de una expresión (el signo "=" no se utiliza para asignación, sino para ecuaciones). La función "**kill**" es usada para eliminar variables que hayan sido asignadas previamente (o todas ellas, usando "**kill(all)**)". Veamos algunos ejemplos:

```
(%i31) a:2;b:4;a*b;
```

$(\%o31)2(\%o32)4(\%o33)8$

```
(%i34) a:4$ b:3.5$ a*b;
```

$(\%o36)14{,}0$

```
(%i37) kill(a)$ a*b;
```

$(\%o38)3{,}5a$

```
(%i39) kill(b)$ a*b;
```

$(\%o40)ab$

1.1.1. Ayuda en línea, ejemplos y demos

Desde la línea de comandos de Maxima podemos obtener ayuda inmediata sobre un comando concreto conociendo su nombre. Esto es cierto para la mayor parte de los comandos, y la sintaxis es:

- **describe(Comando,exact)** o bien **? Comando**, cuando se conoce el nombre exacto.
- **describe(Comando,inexact)** o bien **?? Comando**, cuando no se conoce el nombre exacto.
- **describe("",inexact)**

Los dos primeros son equivalentes y también lo son los dos segundos (el espacio de separación después de "?" es muy importante). El quinto lista todos los comandos para los que existe documentación en línea. De manera que las entradas "? integrate" o "describe (integrate,exact)" dan la información sobre este comando. Lo mismo ocurre con "?? integrat" o "describe(integrat,inexact)".

Para buscar los comandos relacionados con una cadena se puede utilizar: "**apropos("cadena")**", por ejemplo: "**apropos("taylor")**". Sólo se escriben las comillas interiores, en este caso las de taylor, no las exteriores

que enfatizan el nombre del comando. Para algunos comandos puede además obtenerse ejemplos de uso mediante "**example(Comando)**", "**example()**"

El segundo de los ítems de la lista anterior muestra la relación de todos los ejemplos disponibles.

example(sum);

example(complex);

También existe una colección de ejemplos de demostración en forma de batería de sentencias Maxima que se van ejecutando paso a paso una tras otra. En cada una de las etapas es necesario que el operador introduzca el punto y coma de rigor y pulse retorno de carro para pasar a la siguiente. La ejecución de dichos archivos, se realiza con la instrucción "**demo(Archivo.dem)**".

1.2. Constantes, funciones, asignaciones, condicionales y bucles

1.2.1. Constantes matemáticas

Las constantes matemáticas más utilizadas en Maxima se escriben en la forma:

%pi = número pi = 3.14159...
%e = es el número e = 2.71828...
%inf = es el infinito
%i = es la unidad imaginaria = sqrt(-1)
%phi = es el número aureo = (sqrt(5)+1)/2

1.2.2. Funciones matemáticas elementales

Las funciones matemáticas elementales se introducen como sigue:

sqrt(x) = raiz cuadrada de x
exp(x) = exponencial de x
log(x) = logaritmo neperiano de x
log(x)/log(b) = logaritmo en base b de x
sin(x) = seno de x
cos(x) = coseno de x
tan(x) = tangente de x
sec(x) = secante de x
csc(x) = cosecante de x
asin(x) = arco seno de x

acos(x) = arco coseno de x
atan(x) = arco tangente de x
sinh(x) = seno hiperbólico de x
cosh(x) = coseno hiperbólico de x
tanh(x) = tangente hiperbólica de x
asinh(x) = arco seno hiperbólico de x
acosh(x) = arco coseno hiperbólico de x
atanh(x) = arco tangente hiperbólica de x
abs(x) = valor absoluto de x
entier(x) = parte entera de x
round(x) = redondeo de x
max(x1,x2,...), min(x1,x2,...) = máximo, mínimo (x1,x2,...)

1.2.3. Funciones relativas al cálculo con números naturales

Para un número natural n, podemos señalar las siguientes:

n! = factorial de n
primep(n) = nos dice si n es o no primo
oddp(n) = nos dice si n es o no impar
evenp(n) = nos dice si n es o no par
factor(n) = da la descomposición en factores primos de n
divisors(n) = da los divisores de n
remainder(n,m) = da el resto de la división de n entre m
quotient(n,m) = da el cociente de la división de n entre m
binomial(m,n) = es el número combinatorio m sobre n

```
(%i53) 7!;
```

(%o53) 5040

```
(%i54) primep(37);
```

(%o54) $true$

```
(%i55) oddp(27);
```

(%o55) $true$

```
(%i56) oddp(208);
```

(%o56) $false$

```
(%i57) primep(212);
```

$(\%o57) false$

```
(%i58) oddp(208);
```

$(\%o58) false$

```
(%i59) factor(3809);
```

$(\%o59) 13\,293$

```
(%i60) 13*293;
```

$(\%o60) 3809$

```
(%i61) divisors(360);
```

$(\%o61) 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 18, 20, 24, 30, 36, 40, 45, 60, 72, 90, 120, 180, 360$

```
(%i62) remainder(362,7);
```

$(\%o62) 5$

```
(%i63) quotient(362,7);
```

$(\%o63) 51$

```
(%i64) 51*7+5;
```

$(\%o64) 362$

```
(%i65) binomial(362,7);
```

$(\%o65) 152469657972312$

1.2.4. Funciones relativas al cálculo con números complejos

Las operaciones aritméticas entre números complejos se realizan como antes, pero para un número complejo z, podemos utilizar también las siguientes funciones:

rectform(z) = que nos da la forma binómica de z
realpart(z) = da la parte real de z
imagpart(z) = da la parte imaginaria de z

polarform(z) = da la forma polar de z
conjugate(z) = da el conjugado de z
abs(z) = da el módulo de z

```
(%i66) rectform((1+2*%i)^3);
```

$(\%o66) -2\,\%i - 11$

```
(%i67) realpart((1+2*%i)^3);
```

$(\%o67) -11$

```
(%i68) imagpart((1+2*%i)^3);
```

$(\%o68) -2$

```
(%i69) polarform((1+2*%i)^3);
```

$(\%o69) 5^{3/2}\,\%e^{\%i(atan(\frac{sin(3\ atan(2))}{cos(3\ atan(2))}) - \%pi)}$

```
(%i70) rectform(conjugate((1+2*%i)^3));
```

$(\%o70) 2\,\%i - 11$

```
(%i71) abs((1+2*%i)^(-1/2));
```

$(\%o71) \frac{1}{5^{1/4}}$

```
(%i72) polarform((1-%i/2)*(3+4*%i));
```

$(\%o72) \frac{5^{3/2}\,\%e^{\%i(atan(4/3)-atan(1/2))}}{2}$

```
(%i73) rectform(%);
```

$(\%o73) \frac{5^{3/2}\,\%i sin(atan(4/3) - atan(1/2))}{2} + \frac{5^{3/2} cos(atan(4/3) - atan(1/2))}{2}$

```
(%i74) abs((1-%i/2)*(3+4*%i));
```

$(\%o74) \frac{5^{3/2}}{2}$

1.2.5. Operadores lógicos

Los operadores lógicos son:

and = y (conjunción)

not = no (negación)
or = o (disyunción)

Maxima puede averiguar la certeza o falsedad de una expresión mediante el comando

is(expresión) decide si la expresión es cierta o falsa

También podemos hacer hipótesis y olvidarnos de ellas por medio de los comandos

assume(expresión) = suponer que la expresión es cierta
forget(expresión) = olvidar la expresión o hipótesis hecha

```
(%i75) is (7<8 and 9>10);
```

(%o75) *false*

```
(%i75) is (7<8 and 9>10);
```

(%o75) *false*

```
(%i76) is (8>9 or 5<6);
```

(%o76) *true*

```
(%i77) is (x^2-1>0);
```

(%o77) *unknown*

```
(%i78) assume (x>1)$ is(x^2-1>0);
```

(%o79) *true*

```
(%i80) forget(x>1)$ is(x^2-1>0);
```

(%o81) *unknown*

Como se observa en la última salida al olvidarnos de la hipótesis de ser $x > 1$, ya no sabe si es verdadera o falsa la expresión.

1.2.6. Definición de nuevas funciones

Utlizando las funciones y operaciones de Maxima es posible definir nuevas funciones de una o más variables mediante el uso del comando "**:=**", en la forma

"**NombreFunción(x1,x2,...,xn):= expresión**"

Veamos algunos ejemplos de funciones escalares y vectoriales:

```
(%i82) F1(x):=x^3+x^2+x+1;
```

$(\%o82) F1(x) := x^3 + x^2 + x + 1$

Si ahora queremos evaluarla en un punto basta con escribir

```
(%i83) F1(3);
```

$(\%o83) 40$

Para una función escalar de dos variables escribimos

```
(%i84) F2(x,y):=3*x^2-y^5;F2(2,1);
```

$(\%o84) F2(x, y) := 3x^2 - y^5$ $(\%o85) 11$

En tanto que para una función vectorial de tres variables se pone

```
(%i86) F3(x,y,z):=[x-y,x+2*y^2+z,3*y-z^3];F3(1,2,-1);
```

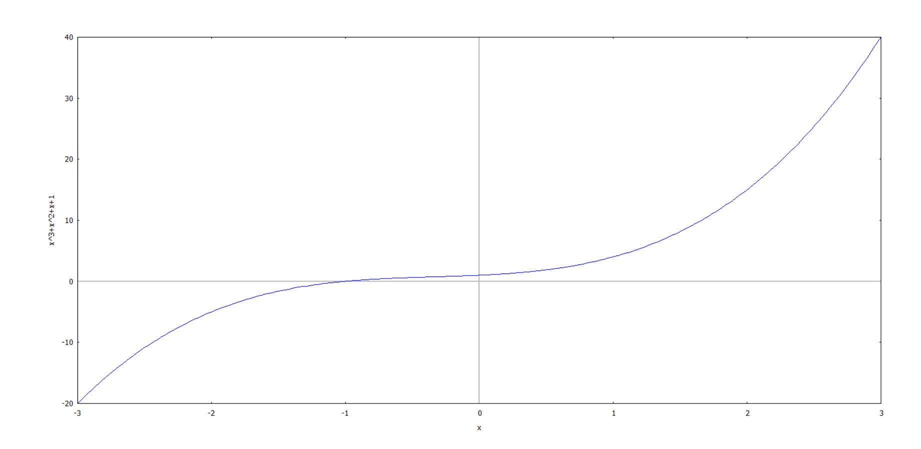
$(\%o86) F3(x, y, z) := [x - y, x + 2y^2 + z, 3y - z^3]$ $(\%o87) [-1, 8, 7]$

Aunque más adelante veremos las representaciones gráficas con más detalle, veamos como representar las funciones escalares de una variable x en un intervalo $[a, b]$ mediante la instrucción "**plot2d(NombreFunción(x),[x,a,b])**".

La representación gráfica la realiza Gnuplot que la muestra en una ventana emergente, la podemos manipular y guardar en distintos formatos.

```
(%i88) plot2d(F1(x),[x,-3,3]);
```

(%o88)



Antes de definir una nueva función conviene comenzar matando la función, por si hubiese sido definida otra con el mismo nombre con anterioridad, lo que se hace con el comando "kill()"; por ejemplo, si quiero definir la función de dos variables $f(x,y) = x^2 - y^2$, haremos lo que sigue

```
(%i89) kill(f)$ f(x,y):=x^2-y^2;
```

(%o90) $f(x,y) := x^2 - y^2$

```
(%i91) f(5,4);
```

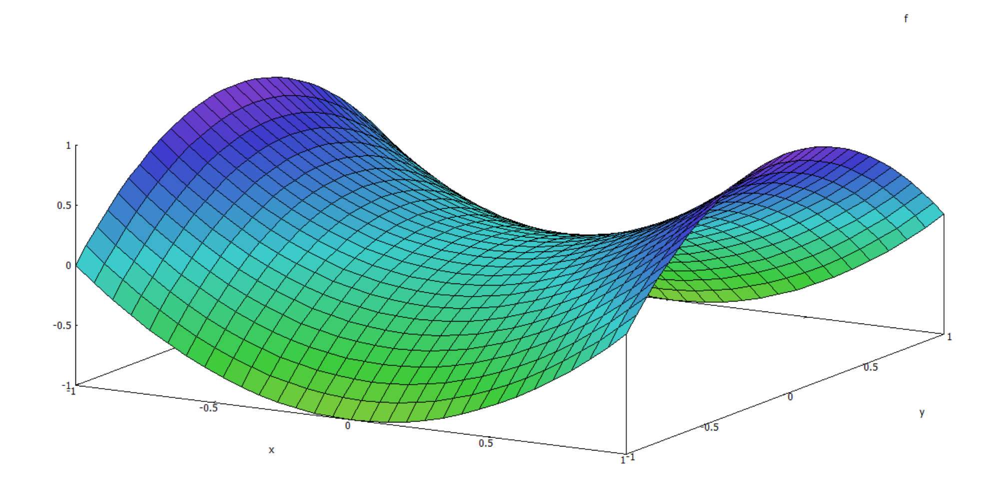
(%o91) 9

Su representación en $[a,b] \times [c,d]$, utiliza el comando

"**plot3d(NombreFunción(x,y),[x,a,b],[y,c,d])**"

```
(%i92) plot3d(f,[x,-1,1],[y,-1,1]);
```

(%o92)



Cuando se define una función mediante el comando ":=" no se desarrolla la expresión que sirve para definirla. Si se desea que la expresión se evalúe hay que emplear el comando (que utilizamos normalmente en la programación):

"**define(NombreFun(x1,x2,...xn), expresión)**"

La diferencia entre ambas formas de definir una función se ilustra en los ejemplos siguientes:

```
(%i93) kill(f,g,h,k)$
       expr : cos(y) - sin(x);
       f(x,y):=expr;
       define(g(x,y),expr);
       f(a,b);
       g(a,b);
       h(x,y):=cos(y) - sin(x);
       h(a,b);
```

(%o94) $cos(y) - sin(x)$
(%o95) $f(x, y) := expr$
(%o96) $g(x, y) := cos(y) - sin(x)$
(%o97) $cos(y) - sin(x)$
(%o98) $cos(b) - sin(a)$
(%o99) $h(x, y) := cos(y) - sin(x)$
(%o100) $cos(b) - sin(a)$

En caso de conveniencia es posible cambiar el nombre de las funciones, tanto las predefinidas en Maxima como las definidas por el usuario mediante la instrucción:
"**alias(NombreNuevo1,NombreOriginal1,NombreNuevo2,NombreOriginal2,...)**"

```
(%i101)alias(sen,sin);
       sen(%pi);
```

(%o101)$[sin]$ (%o102)0

```
(%i103)sen(%pi/6);
```

(%o103)$\frac{1}{2}$

Maxima permite la definición de funciones por recurrencia, por ejemplo si queremos definir el factorial de un número natural o la suma de los n primeros naturales podríamos hacerlo como sigue

```
(%i104)fact(n):= if n=1 then 1 else n*fact(n-1);
```

(%o104)$fact(n) := \ if\ n = 1\ then\ 1\ else\ n\ fact(n-1)$

```
(%i105)fact(4);fact(5);
```

(%o105) 24 (%o106) 120

```
(%i107)sumatorio(n):= if n=1 then 1 else n+sumatorio(n-1);
```

(%o107)$sumatorio(n) :=\ if\ n = 1\ then\ 1\ else\ n + sumatorio(n - 1)$

```
(%i108)sumatorio(5);sumatorio(6);
```

(%o108) 15 (%o109) 21

Para borrar las funciones que hayamos definido podemos utilizar los comandos siguientes:

"**remfunction(f1,f2,...,fn)**" que desliga las definiciones de funciones de sus símbolos f1,f2,...,fn, o bien "**remfunction(all)**" que hace lo mismo con todas las funciones que hayamos definido nosotros.

1.2.7. Asignaciones

Para asignar valores a una constante o variable se utiliza el comando "**:**" Supongamos que queremos calcular $a^2 + b^2$ para $a = 2$ y $b = 3$, entonces comenzamos matando los valores que a y b pudieran tener y luego le asignamos valores como sigue:

```
(%i110)kill(a,b)$ a:2$ b:3$ a^2+b^2;
```

(%o113)13

Cambie las asignaciones anteriores y vuelva a efectuar dicha operación

```
(%i114)kill(a,b)$ a:3$ b:4$ a^2+b^2;
```

(%o117)25

En este otro ejemplo consideramos una función que depende de un parámetro y asignamos valores al parámetro a posteriori: $kill(F1, a)\$\ F1(x) := ax^2 + 1$;. Comprobamos el valor de $F1(3)$; obteniendo $9a + 1$.

El parámetro está presente, por ejemplo al hacer la derivada diff(F1(x),x); que devuelve $2ax$.

Realizamos ahora la asignación de valor al parámetro $a : 2\$$ y volvemos a calcular la derivada comparando el nuevo resultado con el obtenido anteriormente, vemos que el parámetro ha sido sustituido por su valor; es decir, diff(F1(x),x); genera ahora $4x$.

La asignación de valores, como hemos señalado, utiliza "**:**" y no utiliza "=". El símbolo de igualdad se emplea fundamentalmente con las ecuaciones, si bien también aparece en las operaciones de sustitución y evaluación de expresiones.

Por otra parte, la asignación de valores tiene un ámbito de aplicación más amplio que establecer el valor de una constante en una fórmula, tal y como ha sido utilizado más arriba. Por ejemplo, es posible utilizar el comando **:** para definir una función, tal y como hacemos en los ejemplos que siguen. Si desea tener más información sobre otros posibles usos del comando**:** puede consultar el manual en línea de Maxima, en la forma habitual mediante **? :**

```
(%i118)kill(F1,a)$ F1(x):=a*x^2+1;
```

(%o119) $F1(x) := ax^2 + 1$

```
(%i120)F1(3);
```

(%o120) $9a + 1$

```
(%i121)diff(F1(x),x);
```

(%o121) $2ax$

```
(%i122)a:2$ diff(F1(x),x);
```

(%o123) $4x$

Mediante "**:**" se puede definir una función sin declarar la variable por ejemplo con la instrucción: "**F2:diff(F1,x)**" calculamos la derivada de F1 y la asignamos a una nueva función. Ahora, mediante el comando "**subst(x=3,F2)**" se realiza una sustitución formal. Otro código diferente para realizar la evaluación podría ser el que sigue:

kill(all)$ F1:x^2+1; F2:diff(F1,x); **ev(F2, x=3)**;

```
(%i124)kill(all)$ F1:x^2+1;F2:diff(F1,x);subst(x=3,F2);
```

%o1) $x^2 + 1$
(%o2) $2x$
(%o3) 6

```
(%i4)   kill(all)$ F1:x^2+1;
        F2:diff(F1,x);
        ev(F2, x=3);
```

(%o1) $x^2 + 1$
(%o2) $2x$
(%o3) 6

1.2.8. Asumir algo y olvidarse de ello

Cuando Maxima tiene dudas sobre algún dato que puede influir en cual sea su respuesta, antes de darla hace preguntas, por ejemplo si tecleamos “**integrate(exp(a*x),x,0,inf)**” aparecerá lo que sigue

```
(%i4)  integrate(exp(a*x),x,0,inf);
```

Is a positive, negative, or zero?

El comando “**assume(predicado1,predicado2...)**” permite evitar las preguntas dando de antemano las respuestas. Los predicados tan solo pueden ser expresiones formadas con los operadores relacionales:

$=$ igual
$\#$ distinto
$<$ menor que
$\leq$ menor o igual que
$>$ mayor que
$\geq$ mayor o igual que

```
(%i5)  assume(a<0);
       integrate(exp(a*x),x,0,inf);
```

$(\%o5)$ $[a < 0]$
$(\%o6)$ $-\frac{1}{a}$

El comando “**facts()**” permite conocer las suposiciones activas y con el “**forget(predicado)**” se elimina la suposición correspondiente a predicado.

```
(%i7)  facts();
       forget(a<0);
       assume(a>0);
       integrate(exp(a*x),x,0,inf);
```

$(\%o7)$ $[0 > a]$
$(\%o8)$ $[a < 0]$
$(\%o9)$ $[a > 0]$
defint : integral is divergent. −− an error. To debug this try : debugmode(true);

1.2.9. Bloques

Las asignaciones se realizan, en principio, globalmente y afectan a todas las sentencias que se ejecuten después de la asignación y que tengan que ver

con ella. El comando block, entre otras cosas, permite limitar el campo de acción de una asignación a un bloque de código, en la forma

"**block(**$[v_1, ..., v_n], expr_1, ..., expr_m$**)**" o bien

"**block(**$expr_1, ..., expr_m$**)**"

El efecto de la función evalúa las expresiones secuencialmente y devuelve el valor de la última expresión evaluada.

En caso de usarse, las variables $v_1, ..., v_n$ son locales en el bloque y se distiguen de las globales que tengan el mismo nombre. Si no se declaran variables locales entonces se puede omitir la lista. Dentro del bloque, cualquier otra variable distinta de $v_1, ..., v_n$ se considera global. Al iniciarce, block guarda los valores actuales de las variables $v_1, ..., v_n$ y los recupera al finalizar, olvidando en cambio los valores que con carácter local dentro del bloque hayan tenido éstas.

Los comandos block pueden anidarse. Las variables locales se inicializan cada vez que se entra dentro de un nuevo bloque. Las variables locales de un bloque se consideran globales dentro de otro anidado dentro del primero. Si una variable es no local dentro de un bloque, su valor es el que le corresponde en el bloque superior. Este criterio se conoce con el nombre de "alcance dinámico". El valor del bloque es el de la última sentencia o el argumento de la función return, que puede utilizarse para salir del bloque. Veamos un ejemplo de block

```
(%i11) x:4$ y:5$ x*y;
       block([x,y],x:7, y:8, x*y);
       x*y;
```

(%o13) 20
(%o14) 56
(%o15) 20

Como se observa las variables x e y siguen valiendo lo mismo antes y después del block y diferente dentro del mismo por haberlas declarado como variables locales dentro del block.

1.3. Programación con Maxima: Condicionales y bucles

En la programación de algoritmos numéricos es fundamental la utilización de condicionales y bucles, los primeros en Maxima se realizan con la función **if** (si es necesario, acompañada por **else** o **else if**), para más información teclear **? if**. La forma básica sería

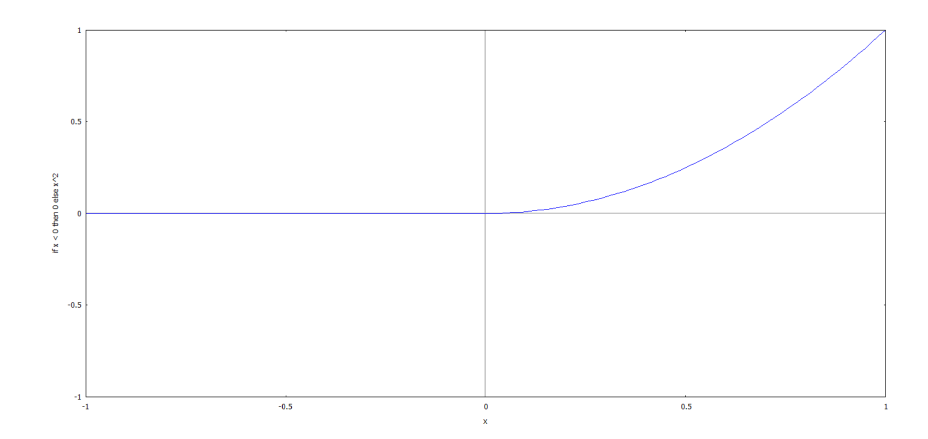
"if condición then acción1 else acción0"

Este comando si condición es verdadera ejecuta acción1 en caso contario es decir si es falsa ejecuta la expresión tras el else o sea acción0.

Veamos algunos ejemplos:

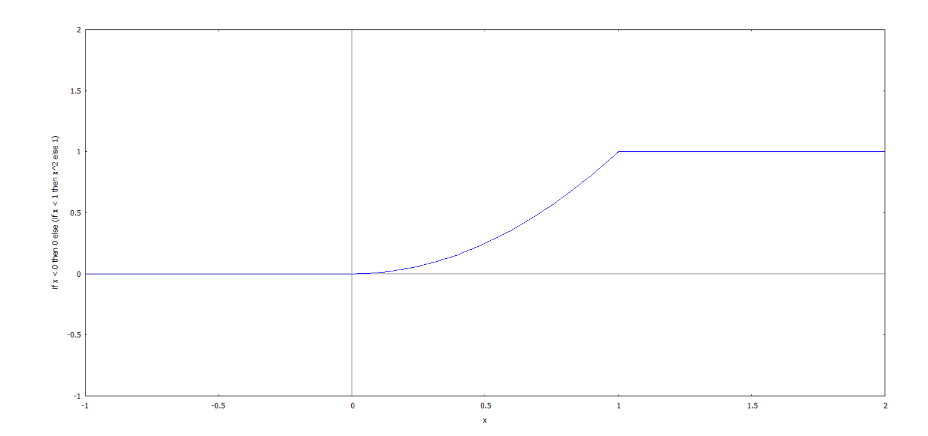
```
(%i17) kill(all)$ F(x):=if x<0 then 0 else x^2;
       plot2d( F(x), [x,-1,1], [y,-1,1] );
```

(%o1)F(x) := if $x < 0$ then 0 else x^2
(%o2)



```
(%i3)  kill(all)$
       F(x):=if x<0 then 0 else (if x<1 then x^2 else 1) ;
       plot2d( F(x), [x,-1,2], [y,-1,2] );
```

(%o1) $F(x) := if\ x < 0\ then\ 0\ else\ (if\ x < 1\ then\ x^2\ else\ 1)$
(%o2)




```
(%i3)  kill(all)$
       F(x):=if (x<0) then 0 elseif (x>0 and x<1) then x^2
       else 1; plot2d( F(x), [x,-1,2], [y,-1,2] );
       plot2d( F(x), [x,-1,2], [y,-1,2] );
```

(%o1) $F(x) := \ if x < 0 \ then \ 0 \ elseif \ (x > 0) \ and \ (x < 1) \ then \ x^2 \ else \ 1$
(%o2) *La misma anterior.*

A continuación mediante if anidados vamos a dar una función que calcula el máximo de tres números dados.

```
(%i3)  f(x,y,z):=if x>=y and y>=z then x elseif y>=x and y>= z
       then y else z;
```

(%o3) $f(x, y, z) := \ if \ (x >= y) \ and \ (y >= z) \ then \ x \ elseif \ (y >= x) \ and \ (y >= z) \ then \ y \ else \ z$

```
(%i4)  f(-1,5,3);
```

(%o4) 5

```
(%i5)  max(-1,5,3);
```

(%o5) 5

Para los bucles disponemos de for, que tiene las siguientes variantes:

for $< var >:< val1 >$ **step** $< val2 >$ **thru** $< val3 >$ **do** $< expr >$
for $< var >:< val1 >$ **step** $< val2 >$ **while** $< cond >$ **do** $< expr >$
for $< var >:< val1 >$ **step** $< val2 >$ **unless** $< cond >$ **do** $< expr >$

El primer for desde el valor inicial val1 de la variable var la va incrementando con paso val2 y siempre que esta sea menor o igual que val3 calcula

expr. En el segundo caso, partiendo del mismo valor inicial y con el mismo incremento, calcula expr mientras se verifique cond. Y en el tercero calcula expr a no ser que se verifique cond.

Cuando el incremento de la variable es la unidad, se puede obviar la parte de la sentencia relativa a step, dando lugar a

for $< var >:< val1 >$ **thru** $< val3 >$ **do** $< expr >$
for $< var >:< val1 >$ **while** $< cond >$ **do** $< expr >$
for $< var >:< val1 >$ **unless** $< cond >$ **do** $< expr >$

Cuando no sea necesaria la presencia de una variable de recuento de iteraciones, también se podrá prescindir de los for, como en:

while $< cond >$ **do** $< expr >$
unless $< cond >$ **do** $< expr >$

Para salidas por pantalla se utilizan los comandos **disp**, **display** y **print**, veremos alguna información adicional en los ejemplos para ampliar información utilizar ?. Veamos unos ejemplos:

```
(%i6)  for k:0  thru 8  do  (angulo:k*%pi/4,print(
       "El  valor del seno de ",angulo, "es ",sin(angulo)));
```

El valor del seno de 0 es 0
El valor del seno de $\pi/4$ es $1/\sqrt{2}$
El valor del seno de $\pi/2$ es 1
El valor del seno de $3\pi/4$ es $1/\sqrt{2}$
El valor del seno de π es 0
El valor del seno de $5\pi/4$ es $-1/\sqrt{2}$
El valor del seno de $3\pi/2$ es -1
El valor del seno de $7\pi/4$ es $-1/\sqrt{2}$
El valor del seno de 2π es 0
(%o6) done

```
(%i7)  for k:0   while  k<3  do  print(k);
```

0
1
2
(%o7) *done*

```
(%i8)  for k:0   unless k>3  do  print(k);
```

0
1

2
3
(%o8)*done*

```
(%i9)  x:1;
       for n:1 thru 9 do x:x*n;
```

(%o9) 1
(%o10)*done*

```
(%i11) x;
```

(%o11) 362880

```
(%i12) x:1$
       n:0$
       while (n<9) do (
         n:n+1,
         x:x*n,
         disp(x)
       )$
```

1
2
6
24
120
720
5040
40320
362880

```
(%i15) x:1$
       n:0$
       while (n<9) do (
         n:n+1,
         x:x*n,
         disp('x)
       )$
```

x
x
x
x

x
x
x
x
x

```
(%i18) x:1$
       n:0$
       while (n<9) do (
         n:n+1,
         x:x*n,
         display(x)
       )$
```

$x = 1$
$x = 2$
$x = 6$
$x = 24$
$x = 120$
$x = 720$
$x = 5040$
$x = 40320$
$x = 362880$

Si queremos sumar los cuadrados de los 100 primeros números naturales con un bucle while ... do escribiremos:

```
(%i21) suma:0;
       indice:0;
       while  indice<=100 do (suma:suma+indice**2,indice:indice+1);
       print(suma)$
```

(%o21) 0
(%o22) 0
(%o23) *done*
338350

Para realizar sumatorios, como no podía ser de otra forma, Maxima tiene implementada la función **sum**, que adopta la forma: **sum(expr,i,i0,i1)** que hace la suma de los valores de expr según el índice i varía de i0 a il. Veamos algún ejemplo:

```
(%i25) sum(i^2,i,1,100);
```

(%o25) 338350

Calcular la suma 8+10+12+...+122

```
(%i26) sum(2*i,i,4,61);
```

(%o26) 3770

```
(%i27) kill(all)$ suma:0$ for i:8 step 2 thru 122 do suma:suma+i;
       print("8+10+12+...+122 = ",suma)$
```

(%o2) *done*
$8 + 10 + 12 + ... + 122 = 3770$

Para realizar productos, Maxima también tiene implementada la función **product**, que adopta la forma: **product(expr,i,i0,i1)** que representa el producto de los valores de expr según el índice i varía de i0 a il. Veamos algún ejemplo:

```
(%i4)  product(x-2*i,i,1,3);
```

(%o4) $(x - 6)(x - 4)(x - 2)$

```
(%i5)  product(x+2*i+1,i,0,3);
```

(%o5) $(x + 1)(x + 3)(x + 5)(x + 7)$

Veamos otro ejemplo, vamos a generar los 20 primeros términos de la sucesión de Fibonacci, dada por $y_1 = 1, y_2 = 1$ e $y_n = y_{n-2} + y_{n-1}$, la definimos por recurrencia y utilizamos un bucle para presentar los resultados de distintas formas (omitimos los números de las salidas):

```
(%i6)  f(n):= if (n=1 or n=2) then(1) else (f(n-2)+f(n-1)) $
       for n:1 thru 20 do disp(f(n))$
```

1
1
2
3
5
8
13
21
34

55
89
144
233
377
610
987
1597
2584
4181
6765

```
(%i8) f(n):= if (n=1 or n=2) then(1) else (f(n-2)+f(n-1)) $
      for n:1 thru 20 do display(f(n))$
```

$f(1) = 1$
$f(2) = 1$
$f(3) = 2$
$f(4) = 3$
$f(5) = 5$
$f(6) = 8$
$f(7) = 13$
$f(8) = 21$
$f(9) = 34$
$f(10) = 55$
$f(11) = 89$
$f(12) = 144$
$f(13) = 233$
$f(14) = 377$
$f(15) = 610$
$f(16) = 987$
$f(17) = 1597$
$f(18) = 2584$
$f(19) = 4181$
$f(20) = 6765$

```
(%i10) f(n):= if (n=1 or n=2) then(1) else (f(n-2)+f(n-1)) $
       for n:1 thru 20 do print("y(",n,")=",f(n))$
```

$y(1) = 1$
$y(2) = 1$
$y(3) = 2$

$y(4) = 3$
$y(5) = 5$
$y(6) = 8$
$y(7) = 13$
$y(8) = 21$
$y(9) = 34$
$y(10) = 55$
$y(11) = 89$
$y(12) = 144$
$y(13) = 233$
$y(14) = 377$
$y(15) = 610$
$y(16) = 987$
$y(17) = 1597$
$y(18) = 2584$
$y(19) = 4181$
$y(20) = 6765$

Más ejemplos de programación con block (en los que utilizamos las variables globales a y x) y bucles, para realizar la suma de los inversos de los i primeros cuadrados y el factorial de i e imprimirlos con un literal y el resultado correspondiente

```
(%i12) f(n):=block(
         a:0,
         x:1,
         for i:1 thru n do print('sum(1/j^2,j,1,i),"=",
         a:a+1/i^2,", ",i,"!=",x:x*i))$

(%i13) f(5);
```

$$\sum_{j=1}^{1} \frac{1}{j^2} = 1, \ 1! = 1$$
$$\sum_{j=1}^{2} \frac{1}{j^2} = \frac{5}{4}, \ 2! = 2$$
$$\sum_{j=1}^{3} \frac{1}{j^2} = \frac{49}{36}, \ 3! = 6$$
$$\sum_{j=1}^{4} \frac{1}{j^2} = \frac{205}{144}, \ 4! = 24$$

$$\sum_{j=1}^{5} \frac{1}{j^2} = \frac{5269}{3600}, \quad 5! = 120$$

(%o13) *done*

```
(%i14) a;
       x;
       kill(a,x)$
       a;
       x;
```

(%o14) $\frac{5269}{3600}$
(%o15) 120
(%o17) a
(%o18) x

Se observa que las variables a y x han sido modificadas dentro del block y conservan su valor cuando se sale del mismo. En cambio en el siguiente bloque las variables [a,x] se declaran como locales y cuando se sale del block tiene los valores que tuvieran inicialmente.

```
(%i19) f(n):=block([a,x],
         a:0,
         x:1,
         for i:1 thru n do print('sum(1/j^2,j,1,i),"=",
         a:a+1/i^2,", ",i,"!=",x:x*i))$
```

```
(%i20) f(5);
```

$$\sum_{j=1}^{1} \frac{1}{j^2} = 1, \quad 1! = 1$$
$$\sum_{j=1}^{2} \frac{1}{j^2} = \frac{5}{4}, \quad 2! = 2$$
$$\sum_{j=1}^{3} \frac{1}{j^2} = \frac{49}{36}, \quad 3! = 6$$
$$\sum_{j=1}^{4} \frac{1}{j^2} = \frac{205}{144}, \quad 4! = 24$$
$$\sum_{j=1}^{5} \frac{1}{j^2} = \frac{5269}{3600}, \quad 5! = 120$$

(%o20) *done*

```
(%i21) a;x;
```

(%o21) a
(%o22) x

1.4. Variables opcionales y funciones útiles para la programación

Entre las muchas variables opcionales de Maxima, repararemos de momento sólo en las siguientes:

- **numer**
- **showtime**
- **powerdisp**

Podemos pedir información sobre ellas tecleando ? seguido de la variable. Por defecto todas ellas tienen asignado el valor false y ver que ocurre si las declaramos como true. Por ejemplo si declaramos "**numer:true**" entonces algunas funciones matemáticas con argumentos numéricos se evalúan como decimales de punto flotante. También hace que las variables de una expresión a las cuales se les ha asignado un número sean sustituidas por sus valores y activa la variable "float". Para volver a su valor predeterminado hay que reiniciar Maxima o bien declararlas como "false". Si hacemos la declaración de la variable "**showtime:true**" se muestra el tiempo de cálculo con la salida de cada expresión, como antes sigue así hasta que se reinicie el programa o se declare como "false". Asimismo, si "**powerdisp**" vale 'true', se muestran las sumas con sus términos ordenados de menor a mayor potencia. Así, un polinomio se presenta como una serie de potencias truncada con el término constante al principio y el de mayor potencia al final. En relación con la variable "showtime" existe la función "**time (%o1, %o2, %o3, ...)**" que devuelve una lista de los tiempos, en segundos, que fueron necesarios para calcular los resultados de las salidas '%o1','%o2', '%o3', ... los tiempos devueltos son estimaciones hechas por Maxima del tiempo interno de computación. La función "time" sólo puede utilizarse para variables correspondientes a líneas de salida; para cualquier otro tipo de variables, "time" devuelve "unknown".

1.5. Aritmética exacta versus aritmética de redondeo

Como dijimos al principio Maxima es un programa de cálculo simbólico y hace las operaciones encomendadas de forma exacta, por ejemplo la suma de fracciones devuelve otra fracción y lo mismo la raíz cuadrada u otras funciones cuyo resultado no sea un entero, a no ser que se le pida mediante **float(número)** o **número, numer**, que dan la expresión decimal de número con 16 dígitos o también con **bfloat(numero)** que da la expresión decimal larga de número acabada con b seguido de un número n, que significa multiplicar por 10^n. Pero trabajar en aritmética exacta no es conveniente en Cálculo Numérico, pues el tiempo empleado en los cálculos aumenta considerablemente, no siendo proporcional al número de operaciones efectuadas y en muchas ocasiones aunque Maxima calcule las operaciones encomendadas no llega a mostrar los resultados. Veamos lo que acabamos de afirmar en los siguientes ejemplos (ejercicio nº 13 de los propuestos al final de esta práctica), en los que se calcula la suma de los inversos de los cuadrados de los $n = 100, 1000, 10000$ primeros números naturales en aritmética exacta y se muestra el tiempo de cálculo empleado en realizar dicha suma:

```
(%i1) numer:false;
```

(%o1) $false$

```
(%i2) showtime:true$ sum (1/i^2,i,1,100);
```

$Evaluation\ took$ 0,0000 $seconds$ (0,0000 $elapsed$)
$Evaluation\ took$ 0,0000 $seconds$(0,0000 $elapsed$)
(%o3)
$$\frac{15895086941330378731122979285\cdots3709859889432834803818131090369901}{9721861444343810305896579766\cdots5746241782720354705517986165248000}$$

```
(%i4) sum (1/i^2,i,1,1000);
```

$Evaluation\ took$ 0,0200 $seconds$ (0,0200 $elapsed$)

(%o4) $$\frac{83545938483149689478187[820 digits]58699094240207812766449}{50820720104325812617835[820 digits]01453118476390400000000}$$

```
(%i5) sum (1/i^2,i,1,10000);
```

$Evaluation\ took$ 0,3500 $seconds$(0,3500 $elapsed$)

(%o5) $$\frac{54714423173933343999582[8648 digits]71491756496677000005717}{33264402841837255560349[8648 digits]95863724851200000000000}$$

```
(%i6) sum (1/i^2,i,1,100000);
```

Evaluation took 20.5300 seconds (20.5300 elapsed) « ¡Expresión excesivamente larga para ser mostrada! »

En general, los ordenadores trabajan en aritmética de redondeo, también llamada de punto **flotante**, con un número fijo k de dígitos; esta aritmética tiene propiedades algo distintas de las habituales, por ejemplo la suma deja de ser asociativa, hay números no nulos que al ser sumados a otro no alteran la suma, se produce la pérdida de cifras significativas al hacer la diferencia de números próximos, etc.; pero se suelen obtener resultados satisfactorios en un tiempo razonable, en general muchísimo menor que en la aritmética exacta y los tiempos empleados tienden a ser proporcionales al número de operaciones realizadas. Recordemos también que por defecto Maxima trabaja en aritmética exacta, de manera que si queremos que lo haga en aritmética de redondeo podemos declararlo mediante **numer:true**. Veamos diversos ejemplos con el anterior sumatorio:

```
(%i7)  numer:true$ sum (1/i^2,i,1,100);
```

Evaluation took 0,0000 *seconds*(0,0000 *elapsed*)
Evaluation took 0,0000 *seconds*(0,0000 *elapsed*)
(%o8) 1,634983900184892

```
(%i9)  sum (1/i^2,i,1,1000);
```

Evaluation took 0,0100 *seconds*(0,0100 *elapsed*)
(%o9) 1,643934566681561

```
(%i10) sum (1/i^2,i,1,10000);
```

Evaluation took 0,0500 *seconds*(0,0500 *elapsed*)
(%o10) 1,644834071848065

```
(%i11) sum (1/i^2,i,1,100000);
```

Evaluation took 0,3700 *seconds* (0,3700 *elapsed*)
(%o11) 1,644924066898243

Se observa que la aritmética de redondeo es mucho más rápida que la exacta. Seguidamente, vamos a aumentar el número de términos en el sumatorio anterior a $1, 2, 4$ u 8 millones para ver como el tiempo de cálculo empleado tiende a ser proporcional al número de operaciones realizadas, primero la ponemos en punto flotante de 16 cifras, luego mostramos el tiempo de cálculo y realizamos los sumatorios pedidos (se trata del ejercicio nº 14 de los propuestos al final de este tema):

```
(%i12) numer:true;
```

Evaluation took 0,0000 *seconds*(0,0000 *elapsed*)
(%o12) *true*

```
(%i14) showtime:true;
```

Evaluation took 0,0000 *seconds*(0,0000 *elapsed*)
(%o14) *true*

```
(%i15) sum (1/i^2,i,1,1000000);
```

Evaluation took 3,3980 *seconds*(3,4000 *elapsed*)
(%o15) 1,644933066848771

```
(%i16) sum (1/i^2,i,1,2000000);
```

Evaluation took 6,5300 *seconds*(6,5400 *elapsed*)
(%o16) 1,644933566848396

```
(%i17) sum (1/i^2,i,1,4000000);
```

Evaluation took 12,5300 *seconds*(12,5400 *elapsed*)
(%o17) 1,644933816848455

```
(%i18) sum (1/i^2,i,1,8000000);
```

Evaluation took 24,6100 *seconds*(24,6200 *elapsed*)
(%o18) 1,644933941848658

Se observa el crecimiento lineal del tiempo de CPU en la aritmética de redondeo (lo que se pone de manifiesto con la gráfica discreta de tiempos empleados frente al número de operaciones), no ocurre lo mismo en la aritmética exacta.

```
(%i19) wxplot2d([discrete, [[1000000, 3.398],[2000000, 6.53],
       [4000000,12.53], [8000000,24.61]]],[style,points]);
```

En el comando **wxplot2d([discrete,** [[**1000000, 3.398**],[**2000000, 6.53**], [**4000000,12.53**], [**8000000,24.61**]]],[**style,points**]), el prefijo wx hace que lo dibuje en la pantalla, el término discrete le informa del tipo de dibujo, los puntos se introducen por pares y style le informa que es un gráfico de puntos.

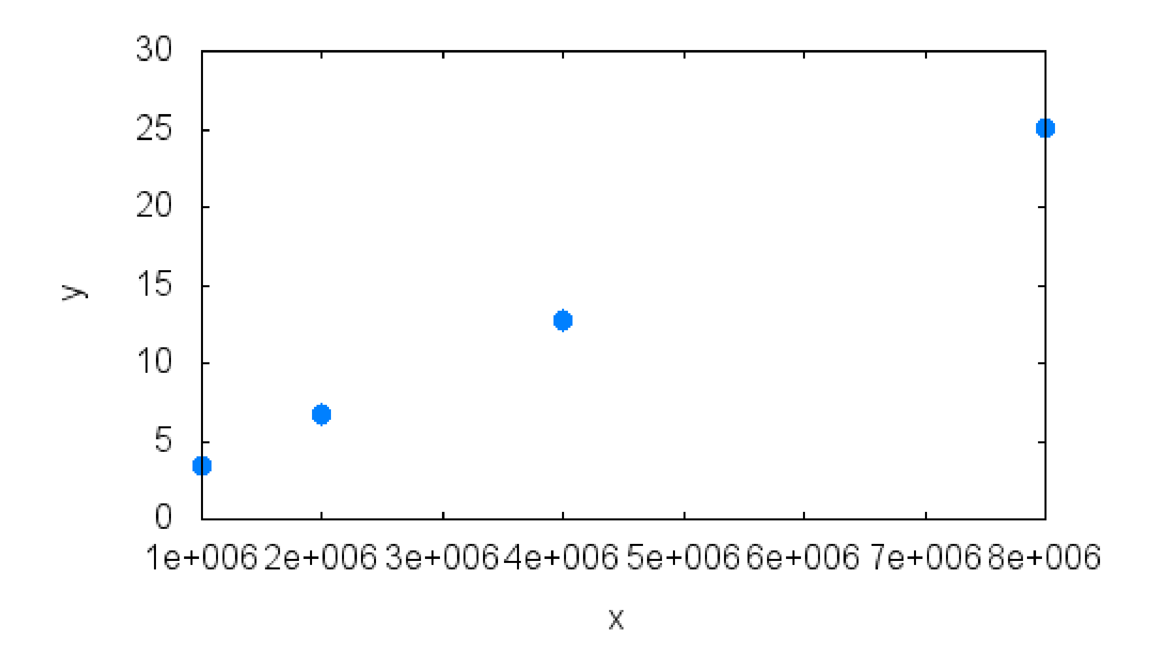


Figura 1.1: Crecimiento lineal del tiempo de CPU en aritmética de redondeo

1.6. Épsilon de máquina

Cuando trabajamos en aritmética de redondeo, conviene recordar que denominamos **épsilon de máquina** ϵ, al número positivo más pequeño que al ser sumado a otro número cualquiera a verifica que $\epsilon + a > a$, de manera que todo número positivo ϵ' menor que ϵ verifica que $\epsilon' + a = a$. Se trata pues de hallar el primer número positivo ϵ tal $1 + \epsilon > 1$, en base 2 será de la forma $0{,}0000000 \cdots 01$, luego una potencia de exponente negativo de 2, hemos de hallar pues el primer número n tal que $1 + 2^{-n} = 1$, en aritmética de punto flotante, en cuyo caso el cero de máquina será $2^{-(n-1)} = 2^{(-n+1)}$ y puede obtenerse, por ejemplo, con el programa (resuelve el ejercicio nº 15 siguiente):

```
(%i21) kill(all)$ fpprec:20$ n:0$ while (1.0+2^(-n)>1.0)
       do (print(" Para n = ",n, "es 1.0+2^(-",n,") = ",
       bfloat(1+2^(-n))," > 1"),n:n+1)$ print (" El épsilon de
       máquina es 2^(",-n+1,") = ",float(2^(-n+1)))$
```

Salida:
Para n $= 0$ es $1{,}0 + 2^{-0} = 2{,}0b0 > 1$
Para n $= 1$ es $1{,}0 + 2^{-1} = 1{,}5b0 > 1$

Para n = 2 es $1{,}0 + 2^{-2} = 1{,}25b0 > 1$

Para n = 3 es $1{,}0 + 2^{-3} = 1{,}125b0 > 1$

Para n = 4 es $1{,}0 + 2^{-4} = 1{,}0625b0 > 1$

Para n = 5 es $1{,}0 + 2^{-5} = 1{,}03125b0 > 1$

Para n = 6 es $1{,}0 + 2^{-6} = 1{,}015625b0 > 1$

Para n = 7 es $1{,}0 + 2^{-7} = 1{,}0078125b0 > 1$

Para n = 8 es $1{,}0 + 2^{-8} = 1{,}00390625b0 > 1$

Para n = 9 es $1{,}0 + 2^{-9} = 1{,}001953125b0 > 1$

Para n = 10 es $1{,}0 + 2^{-10} = 1{,}0009765625b0 > 1$

Para n = 11 es $1{,}0 + 2^{-11} = 1{,}00048828125b0 > 1$

Para n = 12 es $1{,}0 + 2^{-12} = 1{,}000244140625b0 > 1$

Para n = 13 es $1{,}0 + 2^{-13} = 1{,}0001220703125b0 > 1$

Para n = 14 es $1{,}0 + 2^{-14} = 1{,}00006103515625b0 > 1$

Para n = 15 es $1{,}0 + 2^{-15} = 1{,}000030517578125b0 > 1$

Para n = 16 es $1{,}0 + 2^{-16} = 1{,}0000152587890625b0 > 1$

Para n = 17 es $1{,}0 + 2^{-17} = 1{,}00000762939453125b0 > 1$

Para n = 18 es $1{,}0 + 2^{-18} = 1{,}000003814697265625b0 > 1$

Para n = 19 es $1{,}0 + 2^{-19} = 1{,}0000019073486328125b0 > 1$

Para n = 20 es $1{,}0 + 2^{-20} = 1{,}0000009536743164063b0 > 1$

Para n = 21 es $1{,}0 + 2^{-21} = 1{,}0000004768371582031b0 > 1$

Para n = 22 es $1{,}0 + 2^{-22} = 1{,}0000002384185791016b0 > 1$

Para n = 23 es $1{,}0 + 2^{-23} = 1{,}0000001192092895508b0 > 1$

Para n = 24 es $1{,}0 + 2^{-24} = 1{,}0000000596046447754b0 > 1$

Para n = 25 es $1{,}0 + 2^{-25} = 1{,}0000000298023223877b0 > 1$

Para n = 26 es $1{,}0 + 2^{-26} = 1{,}0000000149011611939b0 > 1$

Para n = 27 es $1{,}0 + 2^{-27} = 1{,}0000000074505805969b0 > 1$

Para n = 28 es $1{,}0 + 2^{-28} = 1{,}0000000037252902985b0 > 1$

Para n = 29 es $1{,}0 + 2^{-29} = 1{,}0000000018626451492b0 > 1$

Para n = 30 es $1{,}0 + 2^{-30} = 1{,}0000000009313225746b0 > 1$

Para n = 31 es $1{,}0 + 2^{-31} = 1{,}0000000004656612873b0 > 1$

Para n = 32 es $1{,}0 + 2^{-32} = 1{,}0000000002328306437b0 > 1$

Para n = 33 es $1{,}0 + 2^{-33} = 1{,}0000000001164153218b0 > 1$

Para n = 34 es $1{,}0 + 2^{-34} = 1{,}0000000000582076609b0 > 1$

Para n = 35 es $1{,}0 + 2^{-35} = 1{,}0000000000291038305b0 > 1$

Para n = 36 es $1{,}0 + 2^{-36} = 1{,}0000000000145519152b0 > 1$

Para n = 37 es $1{,}0 + 2^{-37} = 1{,}0000000000072759576b0 > 1$

Para n = 38 es $1{,}0 + 2^{-38} = 1{,}0000000000036379788b0 > 1$

Para n = 39 es $1{,}0 + 2^{-39} = 1{,}0000000000018189894b0 > 1$

Para n = 40 es $1{,}0 + 2^{-40} = 1{,}0000000000009094947b0 > 1$

Para n = 41 es $1{,}0 + 2^{-41} = 1{,}0000000000004547474b0 > 1$

Para n = 42 es $1{,}0 + 2^{-42} = 1{,}0000000000002273737b0 > 1$

Para n = 43 es $1{,}0 + 2^{-43} = 1{,}00000000000011368 68b0 > 1$
Para n = 44 es $1{,}0 + 2^{-44} = 1{,}000000000000056843 4b0 > 1$
Para n = 45 es $1{,}0 + 2^{-45} = 1{,}000000000000028421 7b0 > 1$
Para n = 46 es $1{,}0 + 2^{-46} = 1{,}000000000000014210 9b0 > 1$
Para n = 47 es $1{,}0 + 2^{-47} = 1{,}000000000000007105 4b0 > 1$
Para n = 48 es $1{,}0 + 2^{-48} = 1{,}000000000000003552 7b0 > 1$
Para n = 49 es $1{,}0 + 2^{-49} = 1{,}000000000000001776 4b0 > 1$
Para n = 50 es $1{,}0 + 2^{-50} = 1{,}000000000000000888 2b0 > 1$
Para n = 51 es $1{,}0 + 2^{-51} = 1{,}000000000000000444 1b0 > 1$
Para n = 52 es $1{,}0 + 2^{-52} = 1{,}00000000000000022 2b0 > 1$
El épsilon de máquina es $2^{-52} = 2{,}2204460492503131b - 16$.

Notemos que al poner 1.0 en el programa anterior, este fuerza a Maxima a trabajar en aritmética de redondeo. Otra forma de obtener el épsilon de máquina la da el programa:

```
(%i5)  kill(all)$ epsilon:1.0$
       while ((1+epsilon/2)>1) do(
           epsilon:epsilon/2)$
       print("El épsilon de máquina de Maxima: ",float(epsilon))$
```

Salida: El épsilon de máquina de Maxima: $2{,}2204460492503131 \cdot 10^{-16}$.

Podemos preguntar si el número hallado cumple la definición en la forma

```
(%i6)  is (1+2.2204460492503131*10^-16>1);
```

(%o6) *true*

```
(%i7)  is (1+(2.2204460492503131*10^-16)/2>1);
```

(%o7) *false*

1.7. Programas para el método de bipartición

Veamos un ejemplo de programación con block, creado por J. Rafael Rodríguez Galván, que realiza una utilización más elaborada del comando block para obtener los ceros aproximados de una función continua que cambia de signo en un intervalo. Aparecen en él otros comandos como condicionales y similares propios de programación más avanzada, cuyo significado el lector que conozca el método de bipartición será capaz de comprender.

```
(%i23) biparticion(f_objetivo,a,b,epsilon) :=
       block(
             if( sign(f_objetivo(a)) = sign(f_objetivo(b)) )
             then (print("ERROR, f tiene el mismo signo en a
             y en b"), return(false) )
             else do (c:(a+b)/2,
                     if(abs(f_objetivo(c))<epsilon)
                     then return(c),
                     if(sign(f_objetivo(a))=sign(f_objetivo(c)))
                     then a:c else b:c )
                     )$
```

Este programa aproxima la raíz de una ecuación $f(x) = 0$, cuando f es continua y toma signos opuestos en los extremos, si f toma el mismo signo sale con un mensaje de error. En otro caso, el programa para cuando el valor en un cierto punto medio es menor que el épsilon dado. Veamos seguidamente sendos ejemplos de aplicación.

Utilizando el programa anterior, calculamos ahora el cero de la función $f(x) := (2x)^2 - e^{-x}$, en el intervalo $[0, 1]$, parando el programa cuando el valor de la función $f(x)$ en un cierto punto medio c es menor que 0,01 y mostrando ese punto intermedio c, como valor aproximado de la raíz.

```
(%i24) f(x):=(2*x)^2-exp(-x)$
       biparticion(f,0,1,0.01), numer;
```

$(\%o25)$ 0,40625

Si aplicamos el anterior a la ecuación $x^3 - x - 1 = 0$ en el intervalo $[1, 2]$, tomando épsilon $= 5 \cdot 10^{-16}$, obtendremos:

```
(%i26) kill(f)$ f(x):= x^3-x-1$
       biparticion (f,1,2,5*10^(-16)),numer;
```

$(\%o28)$ 1,324717957244746

Vamos a rehacer el programa anterior para calcular la raíz de $f(x) = 0$ en el intervalo $[a, b]$, deteniendo la ejecución cuando el valor absoluto de la función en un punto medio de algún intervalo generado en el proceso de bipartición sea menor que el épsilon de máquina $(2{,}2204460492503131 \cdot 10^{-16})$(en cuyo caso daremos como salida este punto como la raíz exacta para wxmaxima) o cuando la longitud del enésimo subintervalo sea menor que el épsilon que hayamos escogido (salida que llamaremos la raíz aproximada).

```
(%i29) biparticion1(f_objetivo,a,b,epsilon) :=

       block(
             if(sign( f_objetivo(a)) = sign(f_objetivo(b)) )
              then (print("ERROR, f tiene el mismo signo en a
              y en b"), return(false) )

             else do ( c:(a+b)/2, if( abs(f_objetivo(c))<=
              2.2204460492503131*10^-16)then return((print
              ("La raíz exacta para wxmaxima es c = ",c))),
              if (b-a<epsilon)then return(print("La raíz
              aproximada es ", (a+b)/2," y f(",(a+b)/2,") = ",
              f((a+b)/2)))
                   else if
                   (sign(f_objetivo(a)) = sign(f_objetivo(c)))
                   then a:c else b:c
                     )
            )$
```

```
(%i30) kill(f)$ f(x):= x^3-x-1$
       biparticion1(f,1,2,5*10^(-15)),numer $
```

La raíz aproximada es 1,324717957244745 y $f(1{,}324717957244745) = -2{,}6645352591003757E - 15$.

```
(%i33) kill(f)$ f(x):= x^3-x-1$
       biparticion1(f,1,2,5*10^(-16)),numer $
```

La raíz exacta para wxmaxima es c = 1,324717957244746

```
(%i36) kill(f)$ f(x):= x^3-x-1$
       biparticion1(f,1,2,5*10^(-17)),numer $
```

La raíz exacta para wxmaxima es c = 1,324717957244746

Calculemos ahora la raíz con el comando "**realroots**"(que da aproximaciones racionales de las raíces reales de un polinomio).

```
(%i39) realroots(x^3-x-1=0),numer;
```

(%o39) $[x = 1{,}324717968702316]$

```
(%i40) f(1.324717968702316);
```

(%o40) 4,8862331736287956$E-8$

A continuación, calculando el valor de f en las últimas aproximaciones obtenidas, vemos que nuestro método da mejor solución que el que lleva incorporado Maxima, a no ser que se modifique la variable opcional "**rootsepsilon**", cuyo valor por defecto es $9{,}9999999999999995 \cdot 10^{-8}$.

```
(%i41) f(1.324717957244746);
```

(%o41) 2,2204460492503131$E-16$

1.8. Ejercicios propuestos

Completar con Maxima los ejercicios siguientes:

1. Calcular: $2+1/7+3^4$, $\sqrt{1+\sqrt{4}}+\sqrt{2}$, $\sqrt{2+\sqrt{2+\sqrt{2}}}+64^{1/3}$, $[(\frac{2^5}{3+4/5})^{-3}+6/7]^3$, $\cos(\pi/4)$, $\log(2)$, $\log_{10}(40)$, i^2, $e^{\pi i}$, $\log(-2)$ y $\log(-i)$.
2. Calcular $3\sqrt{27}e^{i(\pi-arctan(13/15))}$, dar sus formas binómica, polar, su módulo y su argumento.
3. Obtener en la misma celda $\log(\pi/4)$, $\log(2)$ y $\sqrt{3}$ en aritmética exacta y en coma flotante.
4. Calcular la $\sqrt{5}$ con 80 dígitos.
5. Obtener el logaritmo en base 7 de 78934123503467896700255432105 89...... en los puntos suspensivos figurará el número de tu DNI.
6. Calcular en aritmética exacta y en coma flotante $arctanh(1)$, el coseno, seno y exponencial de i y $\log(-e)$.
7. ¿Averiguar qué número es mayor 1000^{999} o 999^{1000}?.
8. Define la función $f(x) = \frac{1}{\sqrt{2\pi}}e^{-x^2/2}$ y calcula $f(1)$ tanto en aritmética exacta como en coma flotante (Utiliza la opción del menú - Numérico A real; **observar** que la opción - Conmutar salida numérica hará que todas las salidas sean en coma flotante).
9. Escribir un programa que dados tres números los ordene de mayor a menor. Aplicarlo a los tres siguientes: π, $\frac{73231844868435875}{37631844868435563}$ y $\frac{cosh(3)}{3}$.
10. Utilizando un bucle for, escribe una función que nos de la suma de los cuadrados de los 100 primeros números naturales, hacerlo también con el comando sum.

11. Utilizar un bucle for para poner de manifiesto la acumulación de los errores de redondeo, sumando n veces la fracción $\frac{1}{10} = 0{,}1$ (que vimos era un número periódico en base 2), tomando $n = 1, 10, 100, 1000, 10000, 100000, 1000000$, obtener el error absoluto y relativo y observar la pérdida de cifras significativas de cada resultado.

12. Calcula usando un bucle la $\sum_{n=0}^{50} \frac{n}{n^3+1}$.

13. Averiguar el tiempo de CPU utilizado por tu ordenador para calcular la suma de los n primeros términos de la serie $\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}$ cuando $n = 100, 1000, 10000, 100000$ en aritmética exacta y de coma flotante (Para mostrar tiempo de CPU poner showtime: true).

14. En aritmética de coma flotante comenzando por $n = 1000000$ ve doblando los valores de n en el anterior hasta llegar a $n = 8000000$ y comprueba el crecimiento lineal del tiempo de CPU requerido (no se cumple para aritmética exacta).

15. Se llama épsilon de máquina al menor número positivo ϵ tal que $1+\epsilon > 1$, determinar el epsilón de máquina con Maxima.

Capítulo 2

Introducción a Maxima: Parte II

2.1. Álgebra

2.1.1. Listas

Las listas son objetos básicos a la hora de representar estructuras de datos; de hecho, toda expresión de Maxima se representa internamente como una lista, lo que no es de extrañar habida cuenta de que Maxima está programado en Lisp (List Processing). De ahí que Maxima herede la potencia y versatilidad en el manejo de listas, que se utilizan de forma nativa como bloques de construcción de conjuntos de valores numéricos o simbólicos.

En Maxima las listas se pueden definir como series de expresiones separadas por comas y encerradas entre corchetes. Los elementos de una lista pueden ser también listas, expresiones matemáticas o cadenas de caracteres entre comillas.

En esta sección nos ocuparemos brevemente de la generación y manipulación de listas. Veamos un ejemplo:

```
(%i1)  lista1:[1, 2+1/2,[a,2,c],1+3*x+y^2];
```

(%o1) $[1, \frac{5}{2}, [a, 2, c], y^2 + 3\,x + 1]$

Se puede acceder a los 10 primeros elementos de una lista mediante las funciones "**first**", "**second**",...., "**tenth**", y para el último se accede con "**last**". Por ejemplo

```
(%i2)  second(lista1);last(lista1);
```

(%o2) $\frac{5}{2}$ (%o3) $y^2 + 3x + 1$

También se puede indexar mediante corchetes el elemento enésimo, por ejemplo el tercer elemento de lista se puede pedir en la forma

```
(%i4)  lista1[3];
```

(%o4) $[a, 2, c]$

Con las listas pueden hacerse las operaciones aritméticas básicas de sumar, restar, multiplicar y dividir, término a término, cuando tienen igual longitud. Se puede hacer el producto escalar de vectores (listas), así como potencias y radicales.

```
(%i5)  print("a = ",a:[1,2,3,4,5])$ print("b = ", b:[6,7,3,9,0])$
       print("a + b = ",a+b)$ print("a*b = ",a*b)$
```

$a = [1, 2, 3, 4, 5]$ $b = [6, 7, 3, 9, 0]$ $a + b = [7, 9, 6, 13, 5]$ $a * b = [6, 14, 9, 36, 0]$

```
(%i10) a^2;b/a;
```

(%o10) $[1, 4, 9, 16, 25]$ (%o11) $[6, \frac{7}{2}, 1, \frac{9}{4}, 0]$

```
(%i12) 7*a;
```

(%o12) $[7, 14, 21, 28, 35]$

El producto escalar de los vectores a y b se muestra a continuación

```
(%i13) print("a.b = ",a.b)$
```

(%o13) $a.b = 65$ Diferente de este otro

```
(%i14) print("a*b = ",a*b)$
```

(%o14) $a * b = [6, 14, 9, 36, 0]$

Las listas se pueden manipular de muchas maneras, por ejemplo con la función "**delete**"se puede suprimir un elemento de una lista, en tanto que "**cons**"devuelve la lista que resulta de añadir un elemento al principio de una lista, y se pueden unir listas mediante el comando "**append**"

```
(%i15) ar:delete(3,a);
```

(%o15) $[1, 2, 4, 5]$

```
(%i16) aa:cons(%e,a);
```

(%o16) $[e, 1, 2, 3, 4, 5]$

```
(%i17) append(ar,aa);
```

(%o17) $[1, 2, 4, 5, e, 1, 2, 3, 4, 5]$

```
(%i18) l:[1,2,3,3,3,4,5];
```

(%o18) $[1, 2, 3, 3, 3, 4, 5]$

```
(%i19) lr:delete(3,l);
```

(%o19) $[1, 2, 4, 5]$

```
(%i20) kill(all)$ l1:[1,2,3,4,5]$ l2:[a,b,c,d,e]$ l1+l2;
```

(%o3) $[a + 1, b + 2, c + 3, d + 4, e + 5]$

Podemos construir listas mediante el uso del comando “**makelist**”, formadas por un número finito de expresiones que se ajusten a un término general. Esta función está disponible en wxMaxima a través del menú “**Algebra**”, “**Construir lista...**”. Como se aprecia, sus parámetros son, respectivamente, el término general, la variable utilizada como índice y dos números enteros que expresan el rango en que varía este índice de forma consecutiva.

```
(%i4)  lista:makelist(3*i+1,i,0,10);
```

(%o4) $[1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31]$

```
(%i5)  l:makelist(4*i+1,i,0,15);
```

(%o5) $[1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61]$

```
(%i6)  makelist(7*k^2-1, k, 1, 10);
```

(%o6) $[6, 27, 62, 111, 174, 251, 342, 447, 566, 699]$

```
(%i7)  f1(x):=x^2;
```

(%o7) $\mathrm{f1}(x) := x^2$

```
(%i8)  apply(max,lista);
```

(%o8) 31

```
(%i9)  apply(min,lista);
```

(%o9) 1

```
(%i10) apply("+",lista);
```

(%o10) 176

```
(%i11) apply(f1,lista);
```

Nos da el siguiente mensaje de error:
Too many arguments supplied to $f1(x)$; found: [1,4,7,10,13,16,19,22,25,28,31] - an error. To debug this try: debugmode(true);

Para remediar esta última que da error, está disponible la función "**map**"(en wxMaxima a través del menú "Algebra", "**Distribuir sobre lista**") se utiliza para construir una nueva lista formada por el valor de una función sobre cada uno de los elementos de la lista original.

```
(%i12) map(f1,lista);
```

(%o12) $[1, 16, 49, 100, 169, 256, 361, 484, 625, 784, 961]$

Otras instrucciones relativas a listas son las siguientes:

- **part** busca un elemento dando su posición en la lista
- **reverse** invertir lista
- **sort** ordenar lista
- **flatten** unifica las sublistas en una lista
- **length** longitud de la lista
- **unique** devuelve lista sin elementos repetidos
- **abs** al aplicarlo a lista devuelve la lista de los valores absolutos de los elementos de la lista dada
- **lmax** y **lmin** al aplicarlos a una lista hacen respectivamente lo mismo que apply(max,lista) y apply(min,lista)

```
(%i13) part(lista,5);
```

(%o13) 13

```
(%i14) reverse(lista);
```

(%o14) $[31, 28, 25, 22, 19, 16, 13, 10, 7, 4, 1]$

```
(%i15) length(lista);
```

(%o15) 11

```
(%i16) flatten([[4,5],4,[1,2,3,a]]);
```

(%o16) $[4, 5, 4, 1, 2, 3, a]$

```
(%i17) sort(%);
```

(%o17) $[1, 2, 3, 4, 4, 5, a]$

```
(%i18) unique([1,1,2,3,3,4]);
```

(%o18) $[1, 2, 3, 4]$

```
(%i19) length(%);
```

(%o19) 4

2.1.2. Matrices

Las matrices se introducen en MAXIMA mediante la función "**matrix**", a la que pasamos cada una de sus filas, constituidas por el mismo número de elementos numéricos o simbólicos, escribiendo estos entre corchetes. También cabe la posibilidad de hacerlo a través del menú Álgebra, ofreciéndonos distintas posibilidades.

```
(%i20) A: matrix([1,2,3],[4,5,6],[7,8,9]);
```

(%o20) $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$

Se puede acceder al elemento de la fila 3, columna 1, sin más que poner **A[3,1]**

```
(%i21) A[3,1];
```

(%o21) 7

Se pueden introducir matrices cuyos elementos respondan a una regla, definiendo esta mediante una fórmula y generando la matriz a continuación mediante el comando "**genmatrix**", con el nombre, el número de filas y el de columnas como argumentos, o también mediante el menú "**Algebra, Generar matriz,...**"por ejemplo

```
(%i22) R[i,j]:=i*j-1;
```

(%o22) $R_{i,j} := i\,j - 1$

```
(%i23) B:genmatrix(R,4,6);
```

(%o23) $$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 5 & 7 & 9 & 11 \\ 2 & 5 & 8 & 11 & 14 & 17 \\ 3 & 7 & 11 & 15 & 19 & 23 \end{pmatrix}$$

Se pueden obtener submatrices de una dada mediante el comando:

"**submatrix(l1,l2,...,B,c1,c2,..)**", en el se indica primero las filas a eliminar, segundo el nombre de la matriz y finalmente las columnas a eliminar, por ejemplo si se requiere la submatriz obtenida de la B, suprimiendo las filas 1ª y 3ª, y las columnas 4ª y 5ª, pondríamos

```
(%i24) submatrix(1,3,B,4,5);
```

(%o24) $$\begin{pmatrix} 1 & 3 & 5 & 11 \\ 3 & 7 & 11 & 23 \end{pmatrix}$$

Se puede pedir una fila o columna de una matriz, así como añadir filas o columnas mediante los comandos:

- **row(B,i)**,
- **col(B,j)**,
- **addrow(B,[a,...,z])**
- **addcol(B,[x,...,z])**

```
(%i25) row(B,2);
```

(%o25) $\begin{pmatrix} 1 & 3 & 5 & 7 & 9 & 11 \end{pmatrix}$

```
(%i26) col(B,4);
```

(%o26) $$\begin{pmatrix} 3 \\ 7 \\ 11 \\ 15 \end{pmatrix}$$

```
(%i27) addrow(B,[2,4,6,8,10,12]);
```

(%o27) $$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 5 & 7 & 9 & 11 \\ 2 & 5 & 8 & 11 & 14 & 17 \\ 3 & 7 & 11 & 15 & 19 & 23 \\ 2 & 4 & 6 & 8 & 10 & 12 \end{pmatrix}$$

```
(%i28) addcol(B,[1,3,5,7]);
```

(%o28) $$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 1 \\ 1 & 3 & 5 & 7 & 9 & 11 & 3 \\ 2 & 5 & 8 & 11 & 14 & 17 & 5 \\ 3 & 7 & 11 & 15 & 19 & 23 & 7 \end{pmatrix}$$

Se pueden realizar operaciones algebraicas sobre matrices, con algunas salvedades, así por ejemplo:

Se pueden realizar operaciones algebraicas sobre matrices, con algunas salvedades, así por ejemplo:

El operador (*) se interpreta como producto elemento a elemento.

El operador (.) representa el producto matricial usual

El operador (^) calcula las potencias de los elementos de la matriz dada

El operador (^^) calcula las potencias de la matriz dada

Veamos algunos ejemplos

```
(%i29) m1:matrix([1,2,3],[4,5,6],[7,8,9]);
       m2:matrix([3,2,1],[6,5,4],[9,8,7]);
```

(%o29) $$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

(%o30) $$\begin{pmatrix} 3 & 2 & 1 \\ 6 & 5 & 4 \\ 9 & 8 & 7 \end{pmatrix}$$

```
(%i31) m1*m2;
```

(%o31) $$\begin{pmatrix} 3 & 4 & 3 \\ 24 & 25 & 24 \\ 63 & 64 & 63 \end{pmatrix}$$

```
(%i32) m1.m2;
```

$$(\%o32)\quad \begin{pmatrix} 42 & 36 & 30 \\ 96 & 81 & 66 \\ 150 & 126 & 102 \end{pmatrix}$$

```
(%i33) m1^2;
```

$$(\%o33)\quad \begin{pmatrix} 1 & 4 & 9 \\ 16 & 25 & 36 \\ 49 & 64 & 81 \end{pmatrix}$$

```
(%i34) m1^^2;
```

$$(\%o34)\quad \begin{pmatrix} 30 & 36 & 42 \\ 66 & 81 & 96 \\ 102 & 126 & 150 \end{pmatrix}$$

Que coincide con la matriz producto m1.m1

```
(%i35) m1.m1;
```

$$(\%o35)\quad \begin{pmatrix} 30 & 36 & 42 \\ 66 & 81 & 96 \\ 102 & 126 & 150 \end{pmatrix}$$

Comandos usuales, entre otros, sobre matrices son los siguientes:

- **transpose** da la traspuesta de una matriz
- **determinant** calcula el determinante de una matriz cuadrada
- **rank** da el rango de una matriz
- **invert** da la inversa
- **triangularize** da la matriz triangular superior resultante de aplicar el mátodo de Gauss a una matriz dada
- **eigenvalues** da dos listas, la primera con los valores propios de la matriz y la segunda con sus multiplicidades
- **eigenvectors** da una lista de valores propios junto a una serie de listas de sus autovectores asociados
- **nullspace** da el núcleo de la aplicación lineal definida por esta matriz
- **columnspace** da la imagen de la aplicación lineal definida por esta matriz

- **charpoly(m,x)** da el polinomio característico de la matriz m en la variable x

(%i36) transpose(m1);

(%o36) $\begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$

(%i37) determinant(m1);

(%o37) 0

(%i38) eigenvalues(m1);

(%o38) $[[-\frac{3\sqrt{33}-15}{2}, \frac{3\sqrt{33}+15}{2}, 0], [1, 1, 1]]$

(%i39) eigenvectors(m1);

(%o39) $[[[-\frac{3\sqrt{33}-15}{2}, \frac{3\sqrt{33}+15}{2}, 0],[1,1,1]],[[[1, -\frac{3\sqrt{33}-19}{16}, -\frac{3^{\frac{3}{2}}\sqrt{11}-11}{8}]],$
$[[1, \frac{3\sqrt{33}+19}{16}, \frac{3^{\frac{3}{2}}\sqrt{11}+11}{8}]],[[1, -2, 1]]]]$

(%i40) rank(m1);

(%o40) 2

(%i41) triangularize(m1);

(%o41) $\begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & 0 \end{pmatrix}$

(%i42) charpoly(m1,x);

(%o42) $((5-x)\,(9-x)-48)\,(1-x)-2\,(4\,(9-x)-42)+3\,(32-7\,(5-x))$

(%i43) solve(%,x);

(%o43) $[x = -\frac{3\sqrt{33}-15}{2}, x = \frac{3\sqrt{33}+15}{2}, x = 0]$

(%i44) A1: matrix([1,2,3],[4,5,7],[7,8,9])$ determinant(A1);

(%o45) 6

(%i46) invert(A1);

(%o46) $\begin{pmatrix} -\frac{11}{6} & 1 & -\frac{1}{6} \\ \frac{13}{6} & -2 & \frac{5}{6} \\ -\frac{1}{2} & 1 & -\frac{1}{2} \end{pmatrix}$

```
(%i47) %.A1;
```

(%o47) $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

```
(%i48) nullspace(m1);
```

(%o48) $\mathrm{span}\left(\begin{pmatrix} -3 \\ 6 \\ -3 \end{pmatrix}\right)$

```
(%i49) nullspace(A1);
```

(%o49) $\mathrm{span}\,()$

```
(%i50) columnspace (m1);
```

(%o50) $\mathrm{span}\left(\begin{pmatrix} 1 \\ 4 \\ 7 \end{pmatrix}, \begin{pmatrix} 2 \\ 5 \\ 8 \end{pmatrix}\right)$

```
(%i51) nullspace(m2);columnspace(m2);
```

(%o51) $\mathrm{span}\left(\begin{pmatrix} 3 \\ -6 \\ 3 \end{pmatrix}\right)$

(%o52) $\mathrm{span}\left(\begin{pmatrix} 2 \\ 5 \\ 8 \end{pmatrix}, \begin{pmatrix} 3 \\ 6 \\ 9 \end{pmatrix}\right)$

```
(%i53) nullspace(A1);columnspace(A1);
```

(%o53) $\mathrm{span}\,()$

(%o54) $\mathrm{span}\left(\begin{pmatrix} 1 \\ 4 \\ 7 \end{pmatrix}, \begin{pmatrix} 2 \\ 5 \\ 8 \end{pmatrix}, \begin{pmatrix} 3 \\ 7 \\ 9 \end{pmatrix}\right)$

Ejercicio. Hacer programas para calcular el radio espectral $\rho(A)$, $\| A \|_2$, $\| A \|_1$ y $\| A \|_\infty$ de una matriz dada A.

2.1.3. Expresiones algebraicas

Existe un gran número de funciones relacionadas con la manipulación de ecuaciones y expresiones algebraicas, algunas de las cuales se muestran en el siguiente ejemplo:

- **expand(expr)** para desarrollar expresiones en varios términos
- **ratsimp(expr)** para agrupar en fracciones con denominador común, a veces se usa la siguiente
- **fullratsimp(expr)**, como una reiteración de la anterior
- **factor (expr)** para factorizar expresiones.

En wxMaxima se puede acceder a estas funciones a través del menú “**Simplificar**”.

```
(%i55) (x-2*y)^4;
```

(%o55) $(x-2\,y)^4$

```
(%i56) expand(%);
```

(%o56) $16\,y^4-32\,x\,y^3+24\,x^2\,y^2-8\,x^3\,y+x^4$

```
(%i57) %,x=1/y;
```

sustituye en la expresión anterior x por $1/y$, dando la siguiente salida:

(%o57) $16\,y^4-32\,y^2-\frac{8}{y^2}+\frac{1}{y^4}+24$

```
(%i58) ratsimp(%);
```

(%o58) $\frac{16\,y^8-32\,y^6+24\,y^4-8\,y^2+1}{y^4}$

```
(%i59) factor(%);
```

(%o59) $\frac{(2\,y^2-1)^4}{y^4}$

En los ejemplos anteriores se aprecia que en Maxima se puede **sustituir cualquier expresión, e1, por otra, e2, añadiendo una coma y a continuación una indicación del tipo e1=e2**. También se puede utilizar la función “**subst**”, en la forma “**subst(e2,e1, %)**”. Esta función está disponible en wxMaxima, a través del menú “**Simplificar**”, “**Sustituir...**” y a través del botón [**Sustituir...**]. Las expresiones trigonométricas tienen, además, funciones propias como son:

- **trigexpand** que aplica las propiedades trigonométricas de suma de ángulos para que los argumentos contenidos en las funciones sean lo más simples posibles, y
- **trigreduce** que aplica las propiedades en sentido contrario, de modo que no haya términos que contengan productos de funciones seno o coseno.

Estas funciones están, respectivamente, disponibles en el menú "**Simplificar, Simplificación trigonométrica, Expandir trigonometría...**" y "**Simplificar, Simplificación trigonométrica, Reducir trigonometría...**", así como en los botones [**Expandir (tr)**] y [**Reducir (tr)**].

```
(%i60) sin(x)^3*cos(x-y)+cos(x)^2*sin(x-y);
```

(%o60) $\sin(x)^3 \cos(y-x) - \cos(x)^2 \sin(y-x)$

```
(%i61) trigexpand(%);
```

(%o61) $\sin(x)^3 \left(\sin(x) \sin(y) + \cos(x) \cos(y)\right) - \cos(x)^2 \left(\cos(x) \sin(y) - \sin(x) \cos(y)\right)$

```
(%i62) trigreduce(%);
```

(%o62) $\frac{\sin(y-4x) - \sin(y+2x)}{8} + \frac{-\sin(y+x) - \sin(y-3x)}{4} - \frac{\sin(y-x)}{2} + \frac{3\sin(y) - 3\sin(y-2x)}{8}$

2.1.4. Ecuaciones y sistemas

Aunque profundizaremos algo más en el próximo capítulo, veamos rápidamente algunos comandos de Maxima para resolver ecuaciones o sistemas. En Maxima, el signo = se utiliza para definir ecuaciones. Los sistemas de ecuaciones se definen como listas de ecuaciones, que se escriben entre corchetes, separadas por comas. La función "**solve(ecuaciones,incógnitas)**" devuelve una lista formada por las soluciones de la ecuación (o sistema de ecuaciones) donde la incógnita (o lista de incógnitas) viene dada por incógnitas. En wx-Maxima, se puede acceder a esta función a través del menú **Ecuaciones, Resolver...** o bien del botón [**Resolver...**]

```
(%i63) solve(x^2+x+1=0,x);
```

(%o63) $[x = -\frac{\sqrt{3}\,i+1}{2}, x = \frac{\sqrt{3}\,i-1}{2}]$

```
(%i64) eq1:x+y+z=1 $
       eq2:x-y=0 $
       eq3:2*x-y+z=2 $
       solve([eq1,eq2,eq3],[x,y,z]);
```

(%o67) $[[x = -1, y = -1, z = 3]]$

En el caso de los sistemas lineales es más eficiente utilizar la función: "**linsolve([ecuaciones],[variables])**", veamos nuevamente el ejemplo anterior

```
(%i68) eq:[x+y+z=1, x-y=0,2*x-y+z=2]$
       linsolve(eq,[x,y,z]);
```

(%o69) $[x = -1, y = -1, z = 3]$

Para sistemas algebraicos en general se utiliza la función: "**algsys([ecuaciones],[variables])**", la primera diferencia con solve es que siempre tiene como entrada listas, en otras palabras, **tenemos que agrupar la ecuación o ecuaciones entre corchetes igual que las incógnitas**, la segunda diferencia es que algsys intenta resolver **numéricamente** la ecuación si no es capaz de encontrar la solución exacta. La diferencia entre ambas puede verse en el siguiente ejemplo:

```
(%i70) solve(x^6+x+1=0,x);
```

(%o70) $[0 = x^6 + x + 1]$

```
(%i71) algsys([x^6+x+1=0],[x]);
```

Que da la salida (%o71)

$$[[x = -1{,}0383807544584 61\, i - 0{,}15473514449684],$$
$$[x = 1{,}038380754458461\, i - 0{,}15473514449684],$$
$$[x = -0{,}30050692030955\, i - 0{,}79066718881442],$$
$$[x = 0{,}30050692030955\, i - 0{,}79066718881442],$$
$$[x = 0{,}94540233331126 - 0{,}61183669378101\, i],$$
$$[x = 0{,}61183669378101\, i + 0{,}94540233331126]]$$

Observaciones: Si sólo se requieren raíces reales basta con poner la variable booleana "**realonly**" como "true". Aunque las ecuaciones polinómicas se pueden resolver de manera aproximada mediante los comandos siguientes:

- “**allroots(polinomio)**” y
- “**realroosts(polinomio)**”

que proporcionan soluciones racionales aproximadas de polinomios en una variable, reales en el segundo caso.

```
(%i72) allroots((product(x-i,i,1,20)+10^(-7)*x^19));
```

(%o72) $[x = 1{,}0, x = 2{,}000000000000977, x = 3{,}000000000140009,$
$x = 3{,}999999991528309, x = 5{,}000000222049875, x = 5{,}999992560260244,$
$x = 7{,}000263999613584, x = 7{,}994096489299987, x = 9{,}112077256679337,$
$x = 9{,}570864942275728, x = 1{,}102220454406112\, i + 10{,}92126450508518,$
$x = 10{,}92126450508518 - 1{,}102220454406112\, i,$
$x = 2{,}062169061213381\, i + 12{,}84620554309262,$
$x = 12{,}84620554309262 - 2{,}062169061213381\, i,$
$x = 2{,}69861837443531\, i + 15{,}31474224497542,$
$x = 15{,}31474224497542 - 2{,}69861837443531\, i,$
$x = 2{,}470196797035823\, i + 18{,}15719154663773,$
$x = 18{,}15719154663773 - 2{,}470196797035823\, i,$
$x = 0{,}99919961819071\, i + 20{,}421948379285,$
$x = 20{,}421948379285 - 0{,}99919961819071\, i]$

```
(%i73) algsys([product(x-i,i,1,20)+10^(-7)*x^19=0],[x]);
```

Salida (%o73)

$$[[x = 20{,}421948379285 - 0{,}99919961819071\, i],$$
$$[x = 0{,}99919961819071\, i + 20{,}421948379285],$$
$$[x = 18{,}15719154663773 - 2{,}470196797035823\, i],$$
$$[x = 2{,}470196797035823\, i + 18{,}15719154663773],$$
$$[x = 15{,}31474224497542 - 2{,}69861837443531\, i],$$
$$[x = 2{,}69861837443531\, i + 15{,}31474224497542,]$$
$$[x = 12{,}84620554309262 - 2{,}062169061213381\, i],$$
$$[x = 2{,}062169061213381\, i + 12{,}84620554309262],$$
$$[x = 10{,}92126450508518 - 1{,}102220454406112\, i],$$
$$[x = 1{,}102220454406112\, i + 10{,}92126450508518],$$
$$[x = 9{,}570864847089773], [x = 9{,}112077294685991],$$

$$[x = 7{,}994096812278631], [x = 7{,}000264061262213],$$
$$[x = 5{,}999992560243429], [x = 5{,}0], [x = 4{,}0],$$
$$[x = 3{,}0], [x = 2{,}0], [x = 1{,}0]]$$

Para ampliar información sobre estos comandos y ver sus limitaciones teclear ? seguido del comando correspondiente.

2.2. Cálculo

2.2.1. Límites

El cálculo de límites se realiza con la función "**limit**" en la forma:

- **limit (expr,x,a)** límite de expr cuando x tiende a a
- **limit (expr,x,a,plus)** límite de expr cuando x tiende a a por la derecha
- **limit (expr,x,a,minus)** límite de expr cuando x tiende a a por la izquierda

A su vez a puede ser sustituido por

- **inf** + infinito
- **minf** - infinito

Y el resultado puede tener un valor concreto o bien dar

- **und** indefinido
- **ind** indefinido pero acotado

También podemos calcularlos con el menú "**Análisis-Calcular Límite**".

```
(%i74) limit(n/(2*n+1),n,inf);
```

(%o74) $\frac{1}{2}$

```
(%i75) limit((n-2*n)/(n^3-2),n,inf);
```

(%o75) 0

```
(%i76) limit((n-sin(n))/(n-tan(n)),n,inf);
```

(%o76) *und*

```
(%i77) limit(cos(1/x),x,0);
```

(%o77) *ind*

```
(%i78) limit(sin(x)/(3*x),x,0);
```

(%o78) $\frac{1}{3}$

2.2.2. Derivadas

El cálculo de derivadas se realiza con los comandos:

- **diff(expr,variable)** derivada de expr respecto de variable
- **diff(expr,variable,n)** derivada n-ésima de expr respecto de variable
- **diff(expr,variable1,n1,variable2,n2)** hace la derivada (n1+n2)-ésima de expr respecto de la variable1 n1 veces y respecto de la variable2 n2 veces.

O también a través del menú "**Análisis-Derivar...**"

```
(%i79) func(x):=x^tan(x)+5*sin(log(x))$ diff(func(x),x);
```

(%o80) $\frac{5\cos(\log(x))}{x} + x^{\tan(x)}\left(\frac{\tan(x)}{x} + \log(x)\sec(x)^2\right)$

```
(%i81) kill(f)$ f(x,y):=x^y+y*sin(1/x)$ diff(f(x,y),y);
```

(%o83) $x^y\log(x) + \sin\left(\frac{1}{x}\right)$

```
(%i84) diff(f(x,y),x,2);
```

(%o84) $x^{y-2}\,(y-1)\,y + \frac{2\cos\left(\frac{1}{x}\right)\,y}{x^3} - \frac{\sin\left(\frac{1}{x}\right)\,y}{x^4}$

```
(%i85) diff(f(x,y),y,1,x,1);
```

(%o85) $x^{y-1}\log(x)\,y + x^{y-1} - \frac{\cos\left(\frac{1}{x}\right)}{x^2}$

```
(%i86) diff(f(x,y),y,2);
```

(%o86) $x^y\log(x)^2$

También podemos hallar el jacobiano de una función vectorial de varias variales o el hessiano de una función escalar, en la forma:

- **jacobian**$([xy+z, x^2y-z^3], [x,y,z])$
- **hessian**$(x^2+y^3+z^4, [x,y,z])$

```
(%i87) jacobian([x*y+z,x^2*y-z^3],[x,y,z]);
```

(%o87) $\begin{pmatrix} y & x & 1 \\ 2\,x\,y & x^2 & -3\,z^2 \end{pmatrix}$

```
(%i88) hessian(x^2+y^3+z^4,[x,y,z]);
```

(%o88) $\begin{pmatrix} 2 & 0 & 0 \\ 0 & 6\,y & 0 \\ 0 & 0 & 12\,z^2 \end{pmatrix}$

Si se quiere reutilizar la derivada de una función como una nueva función debe hacerse con el comando "**define(g(x), diff(f(x),x))**", que fuerza a evaluar la derivada.

```
(%i89) define(g(x), diff(func(x),x));
```

(%o89) $\mathrm{g}\left(x\right) := \frac{5\cos\left(\log\left(x\right)\right)}{x} + x^{\tan\left(x\right)}\left(\frac{\tan\left(x\right)}{x} + \log\left(x\right)\,\sec\left(x\right)^2\right)$

Los operadores comilla y dobles comillas tienen un comportamiento muy distinto: **una comilla simple hace que no se evalúe**, en cambio **las dobles comillas obligan a una evaluación** de la expresión que le sigue. Veamos la diferencia cuando aplicamos ambos a una misma expresión:

```
(%i90) 'diff(func(x),x)=''diff(func(x),x);
```

(%o90)

$$\frac{d}{d\,x}\left(5\sin\left(\log\left(x\right)\right) + x^{\tan\left(x\right)}\right) = \frac{5\cos\left(\log\left(x\right)\right)}{x} + x^{\tan\left(x\right)}\left(\frac{\tan\left(x\right)}{x} + \log\left(x\right)\,\sec\left(x\right)^2\right)$$

```
(%i91) 'diff(f(x,y),x)=''diff(f(x,y),x);
```

(%o91) $\frac{d}{d\,x}\left(\sin\left(\frac{1}{x}\right)\,y + x^y\right) = x^{y-1}\,y - \frac{\cos\left(\frac{1}{x}\right)\,y}{x^2}$

```
(%i92) 'diff(f(x,y),x,1,y,1)=''diff(f(x,y),x,1,y,1);
```

(%o92) $\frac{d^2}{d\,x\,d\,y}\left(\sin\left(\frac{1}{x}\right)\,y + x^y\right) = x^{y-1}\log\left(x\right)\,y + x^{y-1} - \frac{\cos\left(\frac{1}{x}\right)}{x^2}$

Veamos algunos comandos usuales en cálculo vectorial, para ello es necesario previamente cargar el paquete "**vect**", mediante la instrucción "**load(vect)**". Entonces podemos calcular:

- El gradiente de una función o campo escalar en la forma
  express(grad(f(x,y,z)); ev(%,diff);
- La divergencia de un campo vectorial en la forma
  express(div([f1(x,y,z),f2(x,y,z),f3(x,y,z)]));ev(%,diff);
- El rotacional de un campo vectorial en la forma
  express(curl([f1(x,y,z),f2(x,y,z),f3(x,y,z)]));ev(%,diff);
- El laplaciano de una función o campo escalar en la forma
  express(laplacian(f(x,y,z)); ev(%,diff);
- El producto vectorial ~

```
(%i93) load(vect);
```

(%o93) C:/PROGRA 2/MAXIMA 1.0-2/share/maxima/
5.28.0-2/share/vector/vect.mac

```
(%i94) grad(log(x*y*z));
```

(%o94) $\mathrm{grad}\left(\log\left(x\,y\,z\right)\right)$

```
(%i95) express(%);
```

(%o95) $[\frac{d}{d\,x}\log\left(x\,y\,z\right),\frac{d}{d\,y}\log\left(x\,y\,z\right),\frac{d}{d\,z}\log\left(x\,y\,z\right)]$

```
(%i96) ev(%,diff);
```

(%o96) $[\frac{1}{x},\frac{1}{y},\frac{1}{z}]$

```
(%i97) express(grad(log(x*y*z))); ev(%,diff);
```

(%o97) $[\frac{d}{d\,x}\log\left(x\,y\,z\right),\frac{d}{d\,y}\log\left(x\,y\,z\right),\frac{d}{d\,z}\log\left(x\,y\,z\right)]$

(%o98) $[\frac{1}{x},\frac{1}{y},\frac{1}{z}]$

```
(%i99) express(grad(x^2*sin(y*x)+z^3)); ev(%,diff);
```

(%o99) $[\frac{d}{d\,x}\left(z^3+x^2\sin\left(x\,y\right)\right),\frac{d}{d\,y}\left(z^3+x^2\sin\left(x\,y\right)\right),\frac{d}{d\,z}\left(z^3+x^2\sin\left(x\,y\right)\right)]$

(%o100) $[2\,x\sin\left(x\,y\right)+x^2\,y\cos\left(x\,y\right),x^3\cos\left(x\,y\right),3\,z^2]$

```
(%i101) express(div([x^3*cos(y*z),x-sin(y*z),x+z^3]));ev(%,diff);
```

(%o101) $\frac{d}{d\,y}\,(x-\sin(y\,z))+\frac{d}{d\,x}\,\left(x^3\cos(y\,z)\right)+\frac{d}{d\,z}\,\left(z^3+x\right)$ (%o102) $-z\cos(y\,z)+3\,x^2\cos(y\,z)+3\,z^2$

```
(%i103) express(laplacian(x^2*sin(y*x)+z^3)); ev(%,diff);
```

(%o103)

$$\frac{d^2}{d\,z^2}\,\left(z^3+x^2\sin(x\,y)\right)+\frac{d^2}{d\,y^2}\,\left(z^3+x^2\sin(x\,y)\right)+\frac{d^2}{d\,x^2}\,\left(z^3+x^2\sin(x\,y)\right)$$

(%o104) $6\,z-x^2\,y^2\sin(x\,y)-x^4\sin(x\,y)+2\sin(x\,y)+4\,x\,y\cos(x\,y)$

```
(%i105) express(grad(x^2*sin(y*x))); ev(%,diff);
```

(%o105) $[\frac{d}{d\,x}\,\left(x^2\sin(x\,y)\right),\frac{d}{d\,y}\,\left(x^2\sin(x\,y)\right),\frac{d}{d\,z}\,\left(x^2\sin(x\,y)\right)]$
(%o106) $[2\,x\sin(x\,y)+x^2\,y\cos(x\,y)\,,x^3\cos(x\,y)\,,0]$

```
(%i107) [2,14,10]~[1,7,5];
```

(%o107) $-[1,7,5]\;[2,14,10]$

```
(%i108) express([-1,2,1]~[2,14,10]);
```

(%o108) $[6,12,-18]$

Los desarrollos de Taylor se realizan con los comandos:

- **taylor** $(<expr>,<x>,<a>,<n>)$
- **taylor** $(<expr>,[<x_1>,<x_2>,...],<a>,<n>)$
- **taylor** $(<expr>,[<x>,<a>,<n>,'asymp])$
- **taylor** $(<expr>,[<x_1>,<x_2>,...],[<a_1>,<a_2>,...],[<n_1>,<n_2>,...])$
- taylor $(<expr>,[<x_1>,<a_1>,<n_1>],[<x_2>,<a_2>,<n_2>],...)$

que dan respectivamente los desarrollos de Taylor de expresión respecto de la variable x alrededor del punto a hasta el orden n, para una o varias variables, para más indicaciones teclear **? taylor**. Veamos un par de ejemplos:

```
(%i110) taylor(sin(x),x,0,11);
```

(%o110)/T/ $x-\frac{x^3}{6}+\frac{x^5}{120}-\frac{x^7}{5040}+\frac{x^9}{362880}-\frac{x^{11}}{39916800}+...$

```
(%i111) taylor(sin(x+y),[x,y],[0,0],10);
```

(%o111)/T/ $x+y-\frac{(x+y)^2}{6}+\frac{(x+y)^5}{120}-\frac{(x+y)^7}{5040}+\frac{(x+y)^9}{362880}+...$

2.2.3. Cálculo integral

En wxMaxima es posible calcular integrales de funciones, tanto definidas como indefinidas, mediante el comando "**integrate**" o bien en el menú "**Análisis - Integrar...**" e introduciendo en la ventana de diálogo la correspondiente función, en la forma:

- **integrate(f(x),x)** que da la primitiva de la función f(x).
- **integrate(f(x),x,a,b)** que da la integral definida de la función f(x) en el intervalo [a,b].

Las integrales múltiples pueden hacerse como una reiteración de integrales simples.

```
(%i112) integrate(x^10/(10+x),x);
```

(%o112) $10000000000\log(x+10)+(63x^{10}-700x^9+7875x^8-90000x^7+1050000x^6-12600000x^5+157500000x^4-2100000000x^3+31500000000x^2-630000000000x)/630$

```
(%i113) integrate(x^10/(10+x),x,0,1);
```

(%o113) $\frac{3150000000000\log(11)-300227066381}{315}-10000000000\log(10)$

```
(%i114) %,numer;
```

(%o114) 0,0083236694335938

```
(%i115) integrate(1/(x^2+1),x,0,inf);
```

(%o115) $\frac{\pi}{2}$

La integración numérica de una función f(x) en un intervalo [a,b] se realiza con alguno de los siguientes comandos:

- **quad_qags(f(x),x,a,b)** para a y b finitos
- **quad_qagi(f(x),x,a,b)** para intervalos ilimitados
- **romberg(f(x),x,a,b)** para a y b finitos

Si se introduce en el menú "**Análisis - Integrar** (y marcamos integración definida e integración numérica)" podemos elegir entre el método **quadpack** o **romberg**; **quad_qags** y **romberg**, se pueden utilizar en intervalos finitos. Para más información sobre estos comandos introducir "? quad_qags" y "? romberg".

```
(%i118) quad_qags(1/(x^2+1), x, 0, 1);
```

(%o118) $[0{,}78539816339745, 8{,}719671245021583\,10^{-15}, 21, 0]$

```
(%i119) romberg(1/(x^2+1),x,0,1);
```

(%o119) 0,7853981595992

```
(%i120) integrate(1/(x^2+1), x, 0, 1);
```

(%o120) $\frac{\pi}{4}$

```
(%i121) %,numer;
```

(%o121) 0,78539816339745

```
(%i122) load(romberg);
```

(%o122) $C : /PROGRA\ 2/MAXIMA\ 1{,}0 - 2/share/maxima/5{,}28{,}0 - 2/share/numeric/romberg.lisp$

```
(%i123) romberg(1/(x^2+1),x,0,1);
```

(%o123) 0,7853981595992

Se puede mejorar esta aproximación cambiando la variable "**rombergtol**", tras haber cargado el paquete "**romberg**" mediante "**load(romberg)**"

```
(%i124) rombergtol:1.0*10^-15;
```

(%o124) $1{,}0000000000000001\,10^{-15}$

```
(%i125) romberg(1/(x^2+1),x,0,1);
```

(%o125) 0,78539816339745

Calculemos integrales impropias con intervalo de integración no acotado, esto se hace sólo con el comando “quad_qagi(f(x),x,a,b)”

```
(%i126) quad_qagi(1/(x^2+1), x, 0, inf);
```

(%o126) $[1{,}570796326794897, 2{,}5777915205990436\, 10^{-10}, 45, 0]$

```
(%i127) romberg(1/(x^2+1), x, 0, inf);
```

(%o127) $\text{romberg}\left(\frac{1}{x^2+1}, x, 0{,}0, \infty\right)$

2.2.4. Ecuaciones diferenciales

Maxima dispone del comando: "**ode2(ecuación diferencial,y,x)**"para resolver algunas ecuaciones de primer o segundo orden; para escribirlas utilizaremos el comando diff precedido de una (’) para evitar que MAXIMA calcule la derivada. Veamos un par de ejemplos (en el primero hallamos la solución general de la ecuación de primer orden $xy^2y' + x^2y = 0$ y en el segundo de la ecuación de segundo orden $x^2y'' + xy' - 2 = 0$):

```
(%i128) edo:x*y^2*'diff(y,x)+x^2*y=0;
        ode2(edo,y,x);
```

(%o128) $x\, y^2 \left(\frac{d}{d\, x}\, y\right) + x^2\, y = 0$

(%o129) $-\frac{y^2}{2} = \frac{x^2}{2} + \%c$

aquí %c es una constante arbitraria.

```
(%i130) edo1:x^2*'diff(y,x,2)+x*'diff(y,x)-2=0; ode2(edo1,y,x);
```

(%o130) $x^2 \left(\frac{d^2}{d\, x^2}\, y\right) + x \left(\frac{d}{d\, x}\, y\right) - 2 = 0$

(%o131) $y = \log(x)^2 + \%k2 \log(x) + \%k1$

en la que %k1 y %k2 son sendas constantes arbitrarias.

Para introducir condiciones iniciales o de contorno se utilizan los comandos:

ic1(soluciondeecuación,x=a,y=b) resuelve problemas de valores iniciales de primer orden

ic2(solucióndeecuación,x=a,y=b,diff(y,x)=c) resuelve problemas de valores iniciales de segundo orden

bc2(solucióndeecuación,x=a,y=b,x=c,y=d) resuelve problemas de contorno para una ecuación de segundo orden

```
(%i134) edo:x*y^2*'diff(y,x)+x^2*y=0$
        ode2(edo,y,x); ic1(%,x=0,y=2);
```

(%o135) $-\frac{y^2}{2} = \frac{x^2}{2} + \%c$

(%o136) $-\frac{y^2}{2} = \frac{x^2 - 4}{2}$

```
(%i137) edo1:x^2*'diff(y,x,2)+x*'diff(y,x)-2=0$ ode2(edo1,y,x);
        ic2(%,x=%e,y=0,diff(y,x)=1);
```

(%o138) $y = \log(x)^2 + \%k2 \log(x) + \%k1$

(%o139) $y = \log(x)^2 - 2\log(x) + 1$

```
(%i140) edo1:x^2*'diff(y,x,2)+x*'diff(y,x)-2=0$ ode2(edo1,y,x);
        bc2(%,x=%e,y=1,x=%e^2,y=2);
```

(%o141) $y = \log(x)^2 + \%k2 \log(x) + \%k1$

(%o142) $y = \log(x)^2 - 2\log(x) + 2$

2.3. Gráficas de funciones

2.3.1. Funciones en 2-D

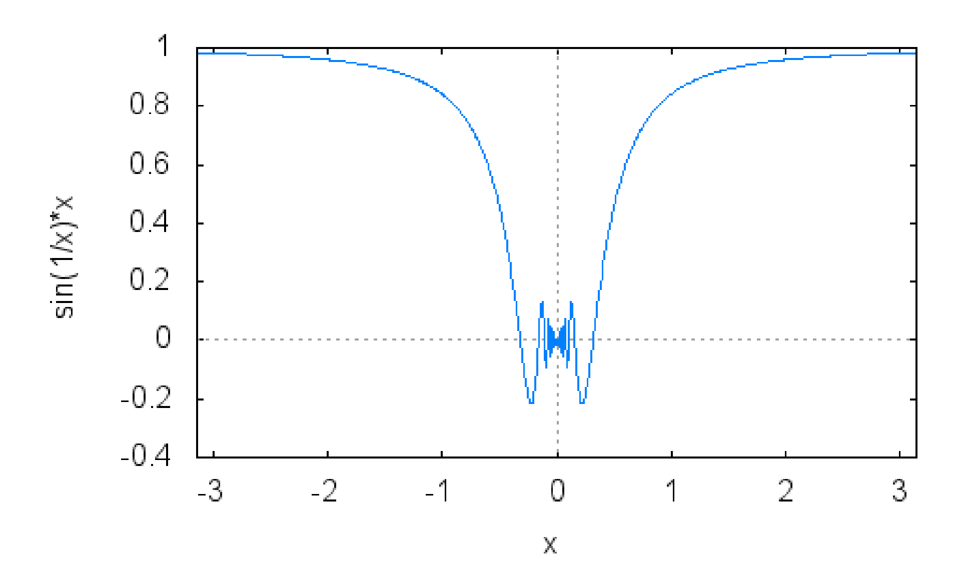
La gráfica de una función de una variable real se hace con el comando **plot2d** que actúa, como mínimo, con dos parámetros: la función (o lista de funciones a representar), y el intervalo de valores para la variable x. Al comando plot2d se puede acceder también a través del menú "**Gráficos - Gráficos 2D**". Por ejemplo: **plot2d(f(x),[x,a,b])** da la gráfica de f(x) en [a, b], en tanto que **plot2d([f1(x),f2(x),...],[x,a,b])** da las gráficas de las funciones f1(x),f2(x),... en [a, b]. Si se le añade el prefijo **wx** delante los presenta en la misma pantalla no en una aparte, como hacen plot o draw.

```
(%i143) plot2d(x*sin(1/x),[x,-%pi,%pi]);
```

SALIDA: plot2d: expression evaluates to non-numeric value somewhere in plotting range. La gráfica correspondiente sale en una pantalla emergente, si la queremos en línea, basta poner el wx delante como viene a continuación.

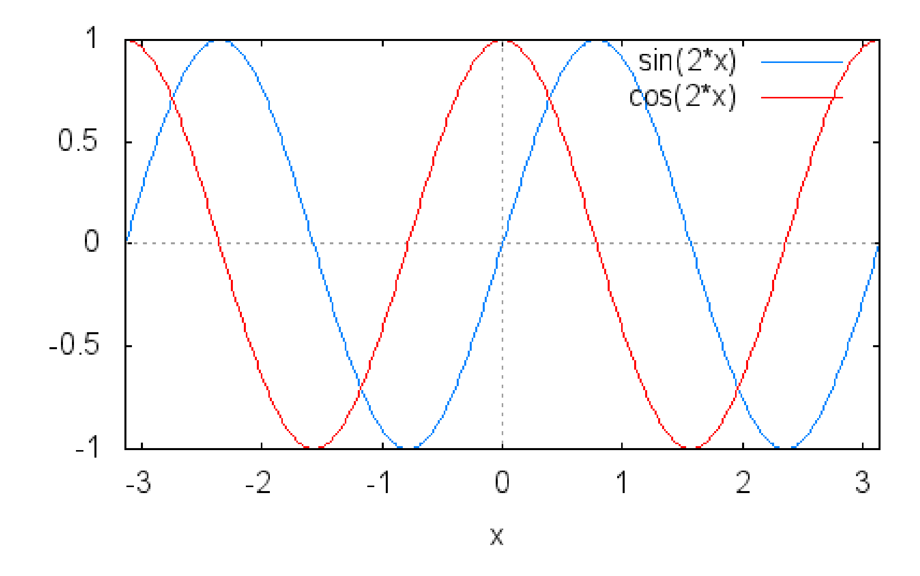
```
(%i144) wxplot2d(x*sin(1/x),[x,-%pi,%pi]);
```

Su salida nos da en pantalla la gráfica de $x \cdot \sin(1/x)$ en el intervalo $[-\pi, \pi]$.

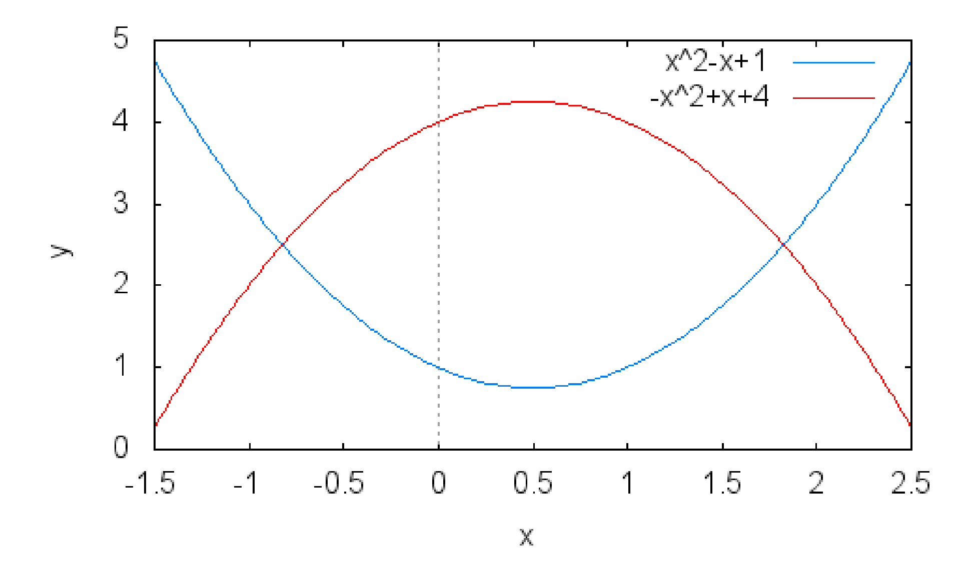



```
(%i146) wxplot2d([sin(2*x),cos(2*x)],[x,-%pi,%pi]);
```

Que nos da las gráficas de $\sin(2x)$ y $\cos(2x)$ en el intervalo $[-\pi, \pi]$.



```
(%i147) plot2d([x^2-x+1,-x^2+x+4], [x,-1.5,2.5], [y,0,5],
        [plot_format, openmath])$
```



Cuando pulsamos el botón **Gráficos 2D** en el menú de Gráficos de wxMaxima, aparece una ventana de diálogo con varios campos que podemos completar o modificar:

a) Expresión(es). La función o funciones que queramos dibujar. Por defecto, wxMaxima rellena este espacio con % para referirse a la salida anterior.

b) Variable x. Aquí establecemos el intervalo de la variable x donde queramos representar la función.

c) Variable y. Ídem para acotar el recorrido de los valores de la imagen.

d) Graduaciones. Nos permite regular el número de puntos en los que el programa evalúa una función para su representación en polares. Veremos ejemplos en la sección siguiente.

e) Formato. Maxima realiza por defecto la gráfica con un programa auxiliar. Si seleccionamos en línea como programa auxiliar wxMaxima, entonces obtendremos la gráfica en una ventana alineada con la salida correspondiente. Hay dos opciones más y ambas abren una ventana externa para dibujar la gráfica requerida: gnuplot es la opción por defecto que utiliza el programa Gnuplot para realizar la representación; también está disponible la opción openmath que utiliza el programa XMaxima. Prueba las diferentes opciones y decide cuál te gusta más.

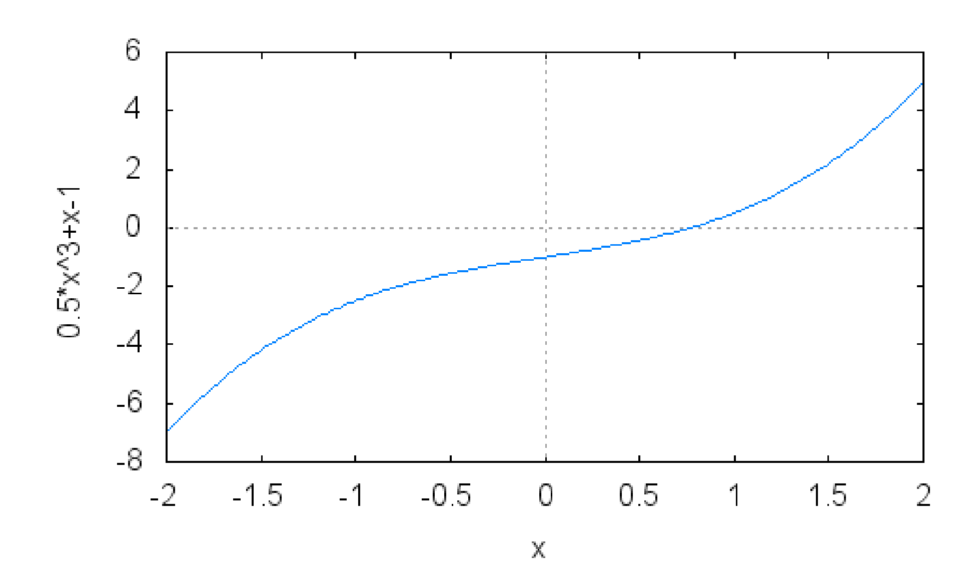
f) Opciones. Aquí podemos seleccionar algunas opciones para que, por ejemplo, dibuje los ejes de coordenadas ("set zeroaxis;"); dibuje los ejes de coordenadas, de forma que cada unidad en el eje Y sea igual que el eje X ("set size ratio 1; set zeroaxis;"); dibuje una cuadrícula ("set grid;") o dibuje una gráfica en coordenadas polares ("set polar; set zeroaxis;"). Esta última opción la comentamos más adelante.

g) Gráfico al archivo. Guarda el gráfico en un archivo con formato Postscript.

Evidentemente, estas no son todas las posibles opciones. La cantidad de posibilidades que tiene Gnuplot es inmensa. Hay opciones para pintar en polares, paramétricas, poligonales, etc. Veamos como funcionan algunas opciones, ponemos el prefijo wx para presentarlas en pantalla.

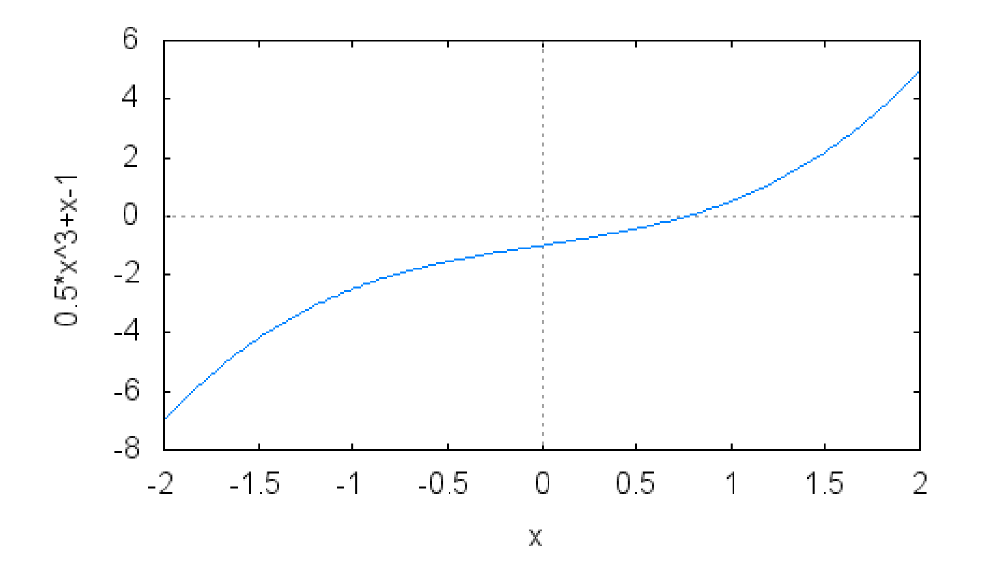
```
(%i149) wxplot2d(0.5*x^3+x-1,[x,-2,2]);
```

(%o149)



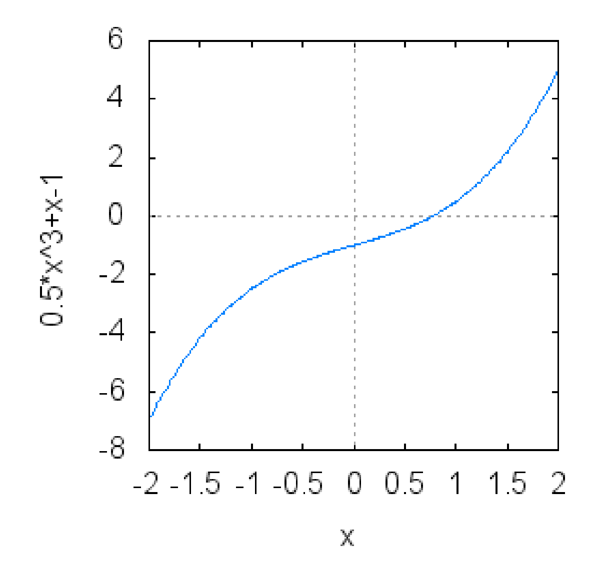

```
(%i150) wxplot2d([0.5*x^3+x-1], [x,-2,2],[y,-8,6],
        [gnuplot_preamble, "set zeroaxis;"])$
```

(%o150)



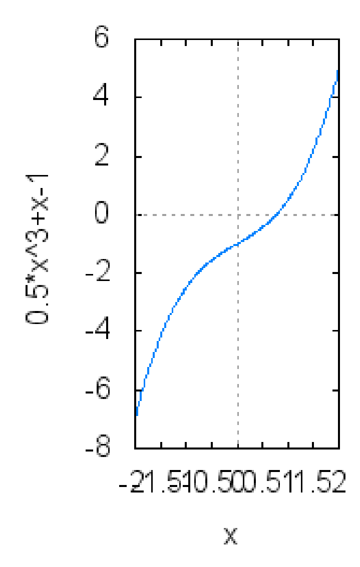

```
(%i151) wxplot2d([0.5*x^3+x-1], [x,-2,2], [y,-8,6],
         [gnuplot_preamble, "set size ratio 1; set zeroaxis;"],
         [nticks,40])$
```

(%o151)



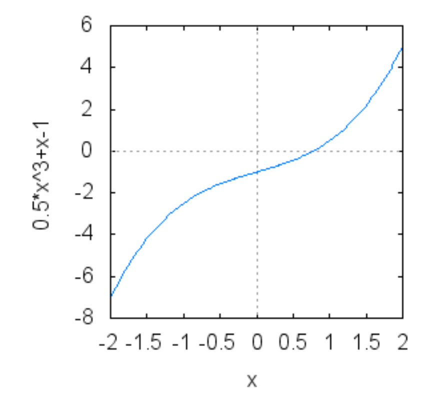
```
(%i152) wxplot2d([0.5*x^3+x-1], [x,-2,2], [y,-8,6],
        [gnuplot_preamble, "set size ratio 2; set zeroaxis;"],
        [nticks,40])$
```

(%o152)



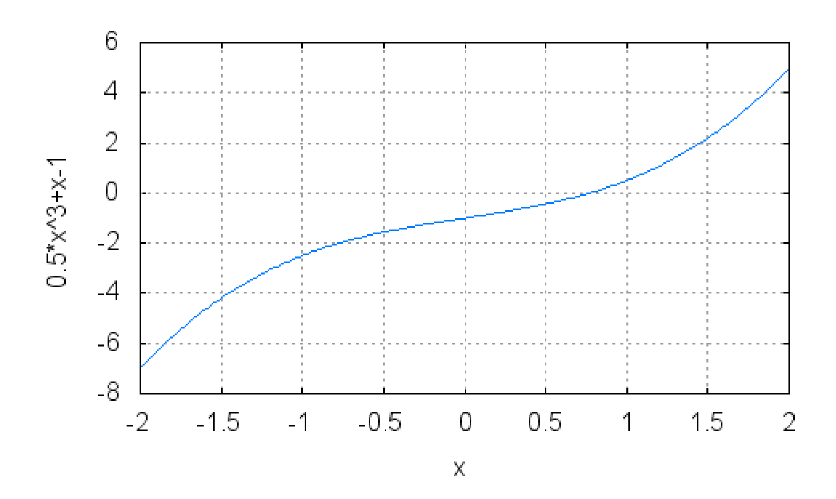

```
(%i153) wxplot2d([0.5*x^3+x-1], [x,-2,2], [y,-8,6],
        [gnuplot_preamble, "set size ratio 1; set zeroaxis;"])$
```

(%o153)



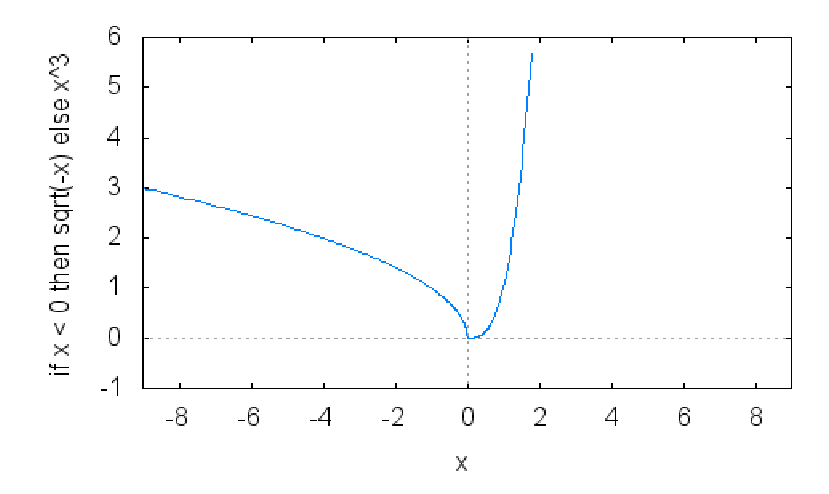
```
(%i154) wxplot2d([0.5*x^3+x-1], [x,-2,2], [y,-8,6],
        [gnuplot_preamble, "set grid;"])$
```

(%o154)



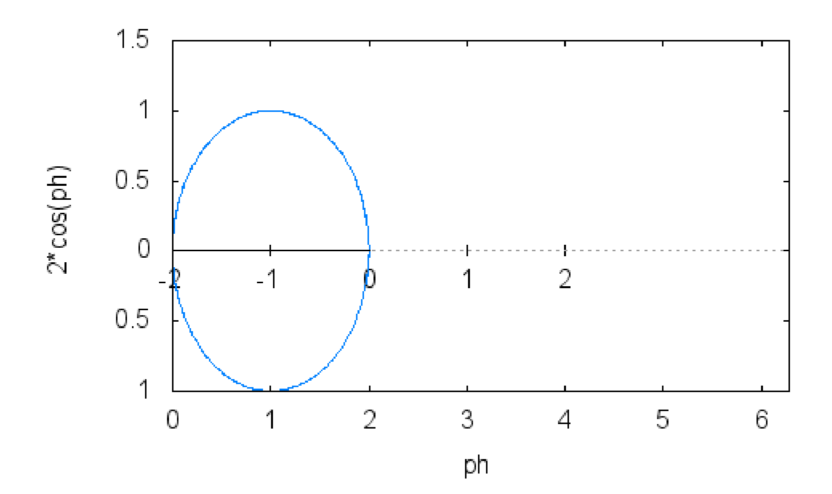

```
(%i155) f(x):= if x<0 then sqrt(-x) else x^3$  wxplot2d(f(x),
        [x,-9,9], [y,-1,6], [gnuplot_preamble, "set zeroaxis;"]);
```

plot2d: some values were clipped.
(%o156)



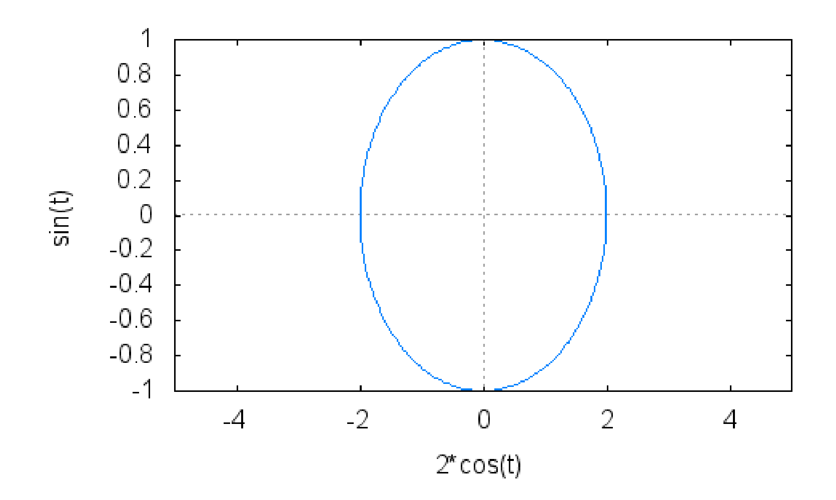
```
(%i157) wxplot2d([2*cos(ph)], [ph,0,2*%pi],
         [gnuplot_preamble, "set polar; set zeroaxis;"])$
```

(%o157)



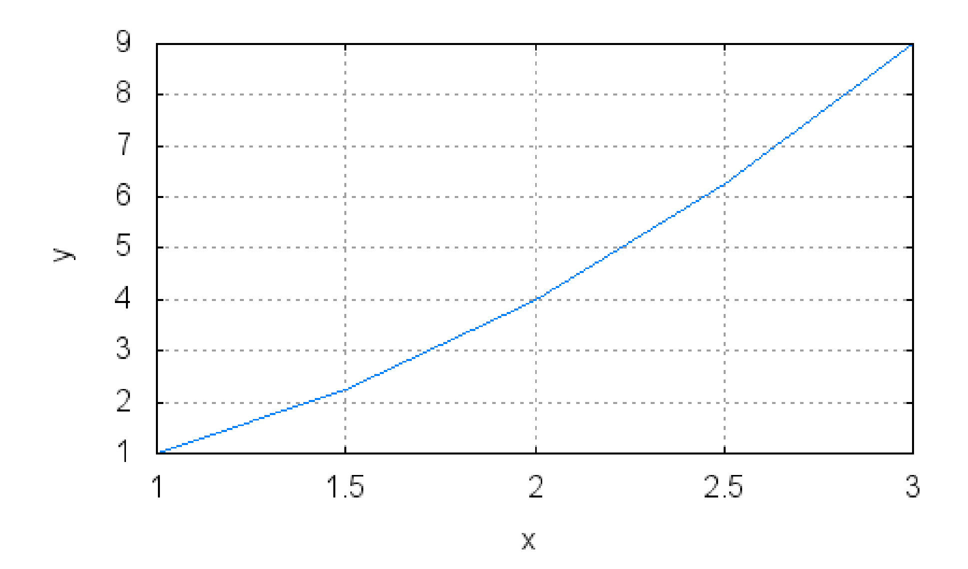

```
(%i158) wxplot2d([['parametric, 2*cos(t), sin(t), [t, -%pi, %pi],
        [nticks, 300]]], [x,-5,5], [gnuplot_preamble, "set zeroaxis;"])$
```

(%o158)



```
(%i159) wxplot2d([['discrete, [1, 1.5, 2, 2.5, 3],
        [1, 2.25, 4,6.25,9]]], [x,1,3], [y,1,9],
        [gnuplot_preamble, "set grid;"])$
```

(%o159)



2.3.2. Funciones en 3-D

También podemos representar funciones de dos variables de forma similar a como hemos representado las de una. Aunque ahora debemos utilizar el comando "**plot3d**" en lugar de plot2d, pero igual que en el caso anterior, son obligatorios la función o funciones a representar y el dominio de definición de las variables, la forma ahora es la siguiente:

"**plot3d(f(x,y),[x,a,b],[y,c,d])**" y da la gráfica de $f(x, y)$ en $[a, b] \times [c, d]$ que se puede girar sin más que pinchar sobre ella con el cursor y desligarlo.

Si se requiere el dibujo en la misma ventana hay que añadir wx delante en la forma:
"**wxplot3d(f(x,y),[x,a,b],[y,c,d])**"

Como siempre, se puede acceder al comando plot3d a través del menú:
"**Gráficos-Gráficos 3D**" y hay diversas opciones con las que podemos experimentar. Veamos algunos ejemplos:

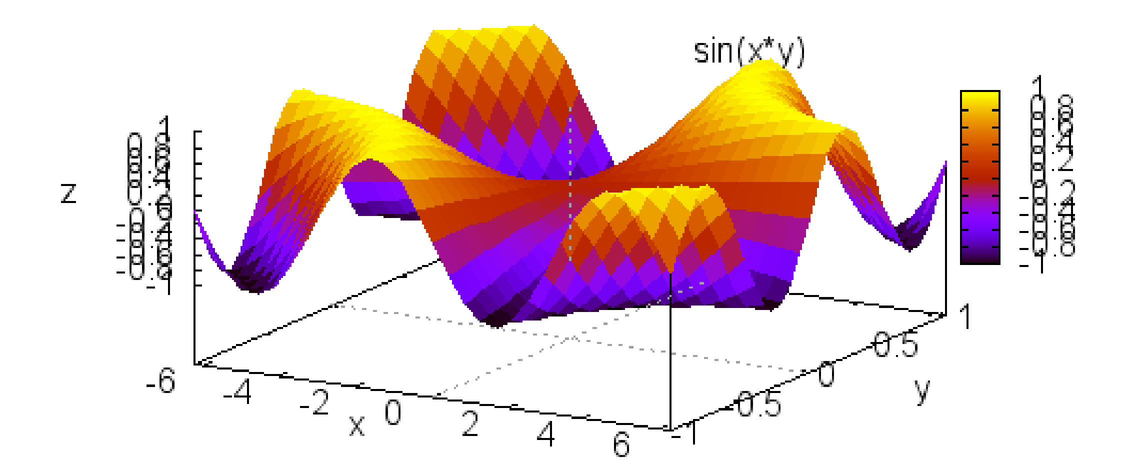
```
(%i160) plot3d(sin(x*y),[x,-2*%pi,2*%pi],[y,-1,1]);
```

SALIDA:La gráfica correspondiente sale en una pantalla emergente, si la queremos en línea, basta poner el wx delante como viene a continuación.

(%o160)

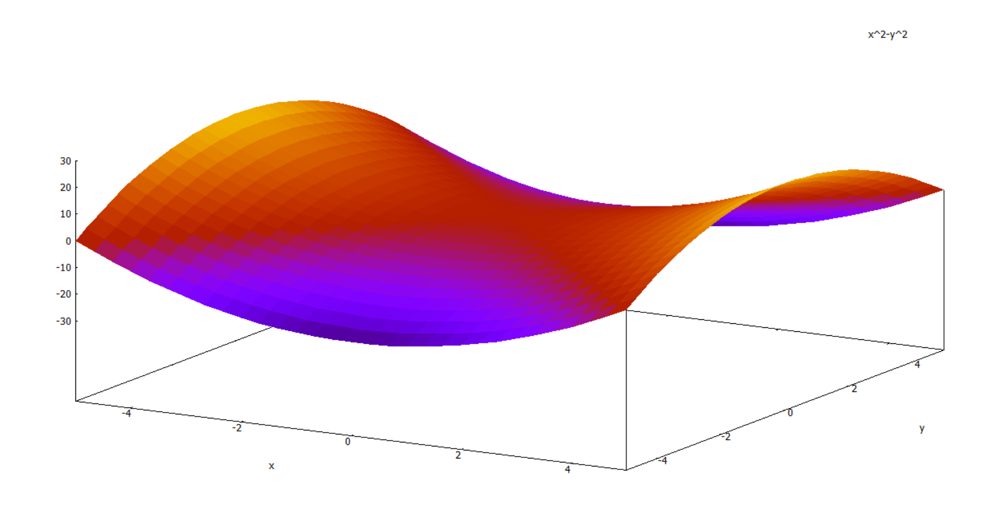
```
(%i161) wxplot3d(sin(x*y),[x,-2*%pi,2*%pi],[y,-1,1]);
```

(%t161)



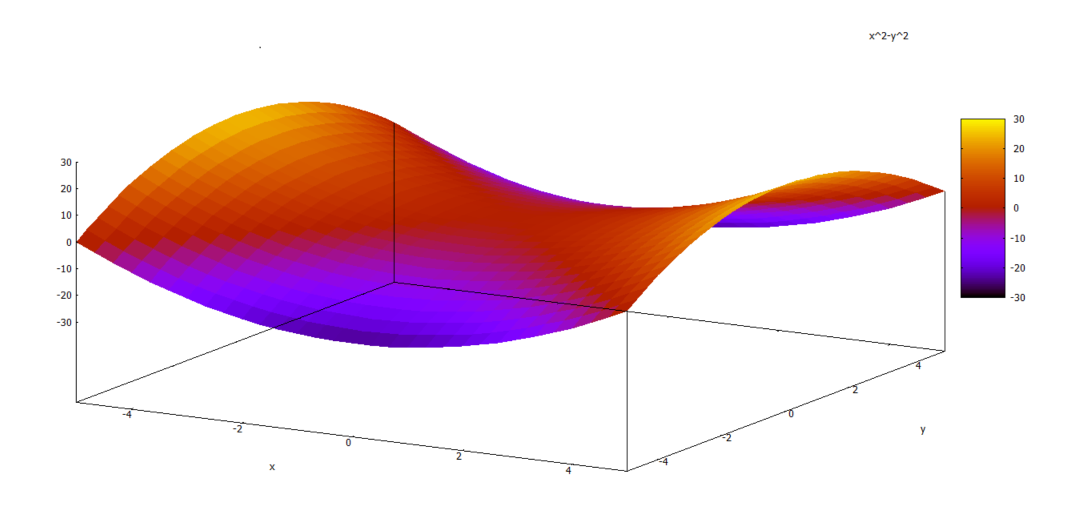
(%o161)

```
(%i162) plot3d(x^2-y^2, [x,-5,5], [y,-5,5], [plot_format,gnuplot],
        [gnuplot_preamble, "set pm3d at s; unset surf; unset colorbox"])$
```



(%o162)

```
(%i163)plot3d(x^2-y^2, [x,-5,5], [y,-5,5], [plot_format,gnuplot],
        [gnuplot_preamble, "set hidden3d"])$
```



(%o163)

2.3.3. Gráficas con draw

El módulo "**draw**" permite dibujar gráficos en 2 y 3 dimensiones con relativa comodidad, para ello se comienza cargándolo mediante la orden "**load(draw)**"

Luego, se introduce alguna de las órdenes

- **gr2d(opciones, objeto gráfico,...)** para un gráfico bidimensional
- **gr3d(opciones, objeto gráfico,...)** para un gráfico tridimensional
- **draw(opciones, objeto gráfico,...)** dibuja un gráfico
- **draw2d(opciones, objeto gráfico,...)** dibuja gráfico bidimensional
- **draw3d(opciones, objeto gráfico,...)** dibuja gráfico tridimensional

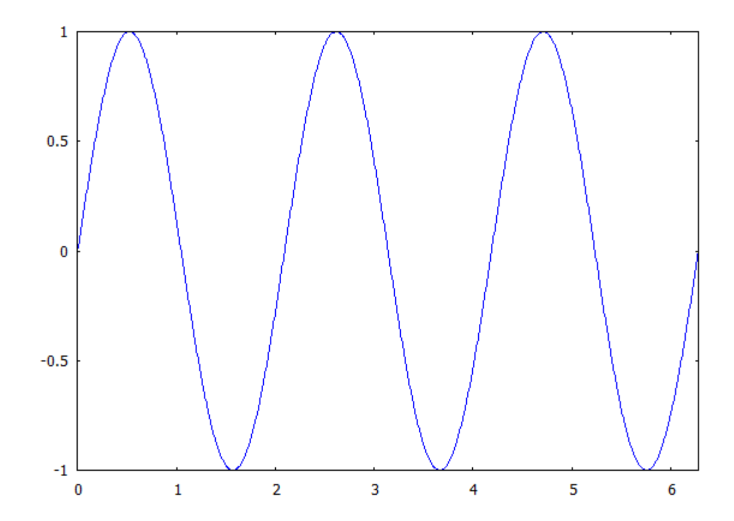
Una vez cargado el módulo draw, podemos utilizar las órdenes draw2d y draw3d para dibujar gráficos en 2 y 3 dimensiones o draw. Un gráfico está compuesto por varias opciones y el objeto gráfico que queremos dibujar. Las opciones son numerosas y permiten controlar prácticamente cualquier aspecto imaginable. Aquí comentaremos algunas de ellas pero conviene recurrir a la ayuda del programa para ampliar conocimientos. En segundo lugar aparece el objeto gráfico que puede ser de varios tipos aunque los que más usaremos son quizás **explicit** y **parametric**. Para dibujar un gráfico tenemos dos posibilidades:

1. Si tenemos previamente definido el objeto, draw(objeto), o bien,
2. draw2d(definición del objeto) (o bien draw3d(definición del objeto)) si lo definimos en ese momento para dibujarlo.

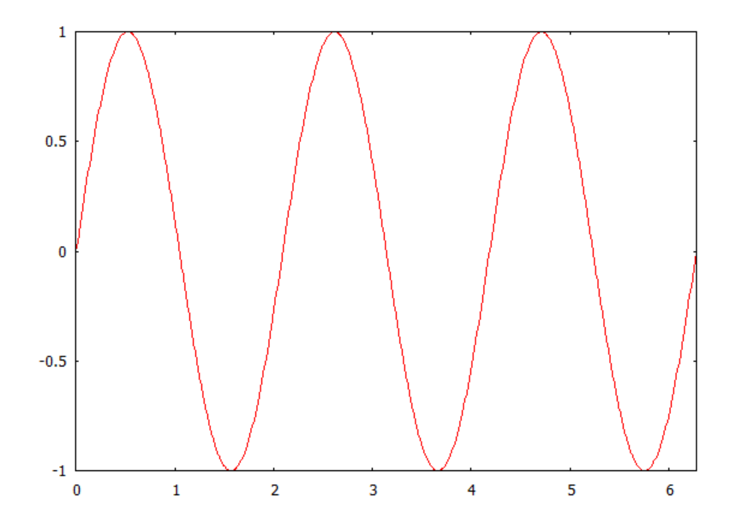
Veamos diversos ejemplos y observemos las diferentes salidas.

```
(%i164) load(draw)$
        objeto:gr2d(
        color=blue,
        nticks=60,
        explicit(sin(3*t),t,0,2*%pi)
        )$ draw(objeto);
```

(%o166)



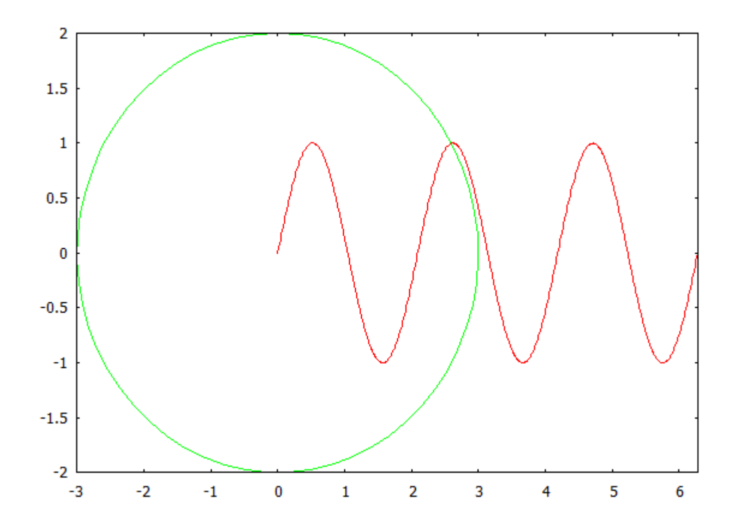

```
(%i167) draw2d(color=red,explicit(sin(3*t),t,0,2*%pi))$
```



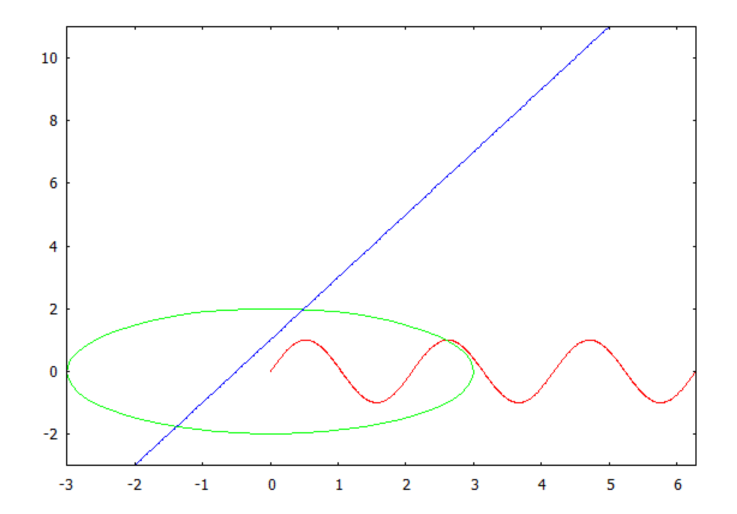
(%o167)

```
(%i168) draw2d(color=red,explicit(sin(3*t),t,0,2*%pi),
        color=green,nticks=60,parametric(3*sin(t),2*cos(t),t,0,2*%pi));
```

(%o168)

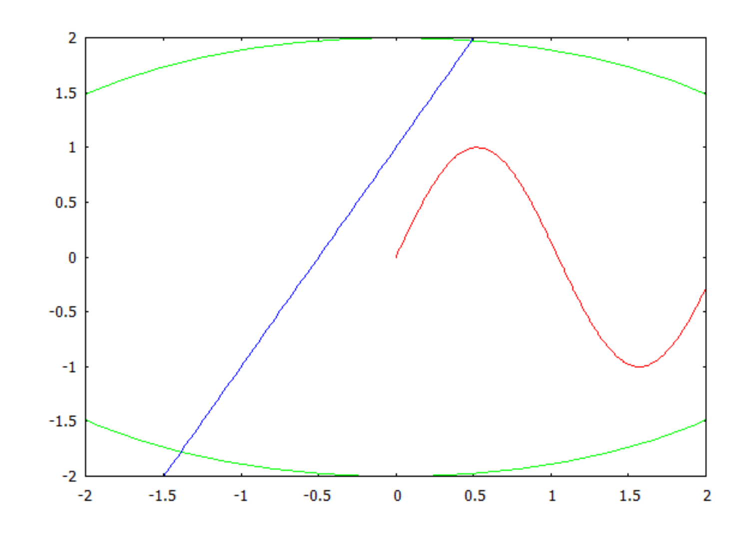



```
(%i169) draw2d(color=red,explicit(sin(3*t),t,0,2*%pi),
        color=green,nticks=60,parametric(3*sin(t),2*cos(t),t,0,2*%pi),
        color=blue,explicit(2*x+1,x,-2,5));
```



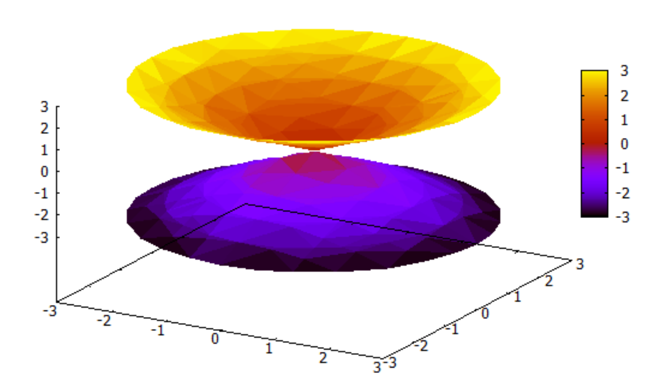
(%o169)

```
(%i170) draw2d(color=red,explicit(sin(3*t),t,0,2*%pi),
        color=green,nticks=60,parametric(3*sin(t),2*cos(t),t,0,2*%pi),
        color=blue,explicit(2*x+1,x,-2,5),xrange=[-2,2],yrange=[-2,2]);
```



(%o170)

```
(%i171) draw3d(
        surface_hide=true,
        enhanced3d=true,
        implicit(x^2+y^2=z^2,x,-3,3,y,-3,3,z,-3,3));
```

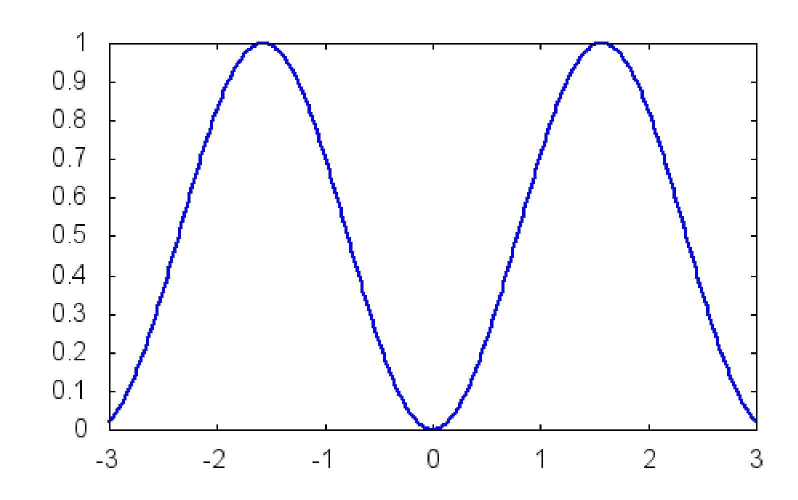


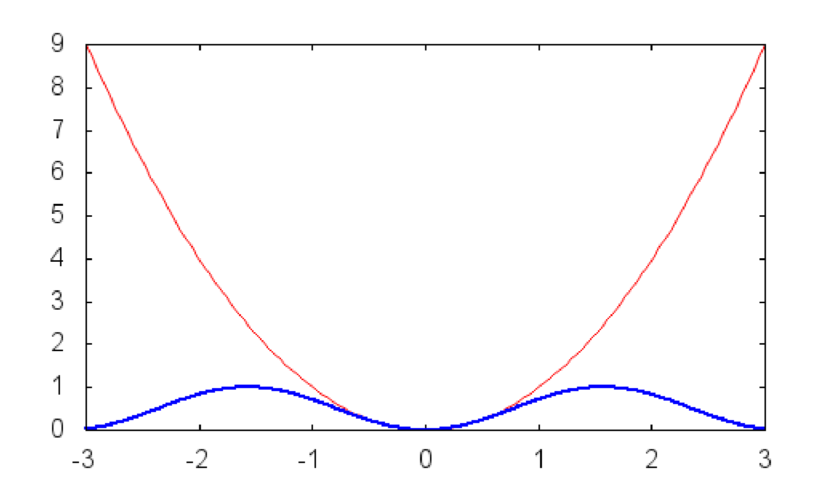
(%o171)

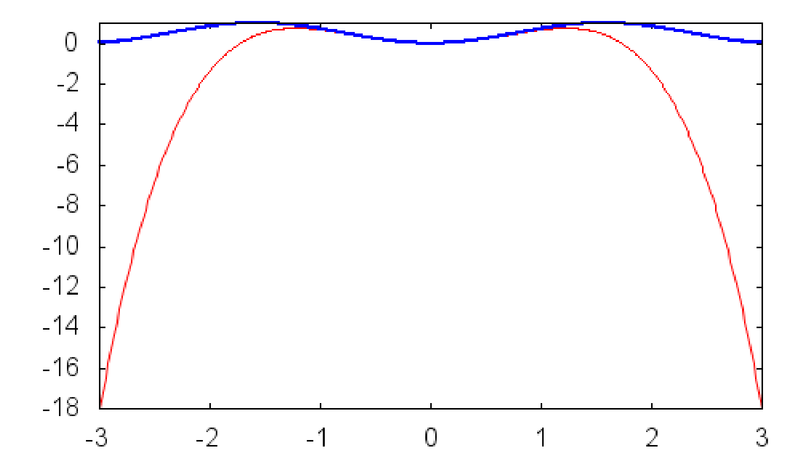
Veamos algunas animaciones con **"for"** o con el comando **"with_slider"**

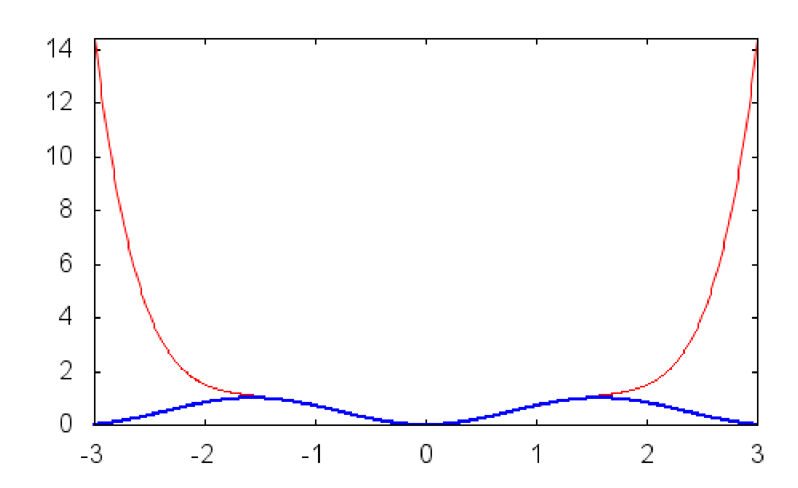
```
(%i172)load(draw); kill(f)$ f(x):=sin(x)^2;for k:1 step 2 thru 18 do
       wxdraw2d(
       color=red,
       explicit(taylor(f(x),x,0,k),x,-3,3),
       color=blue,
       line_width=2,
       explicit(f(x),x,-3,3));
```

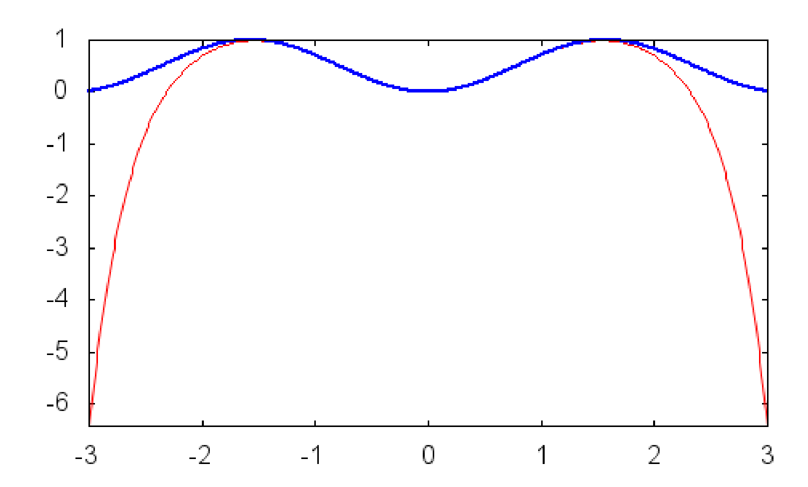
(%o172) $C : /PROGRA\,2/MAXIMA\,1{,}0-2/share/maxima/5{,}28{,}0-2/share/draw/draw.lisp$ (%o174) $f(x) = \sin(x)^2$

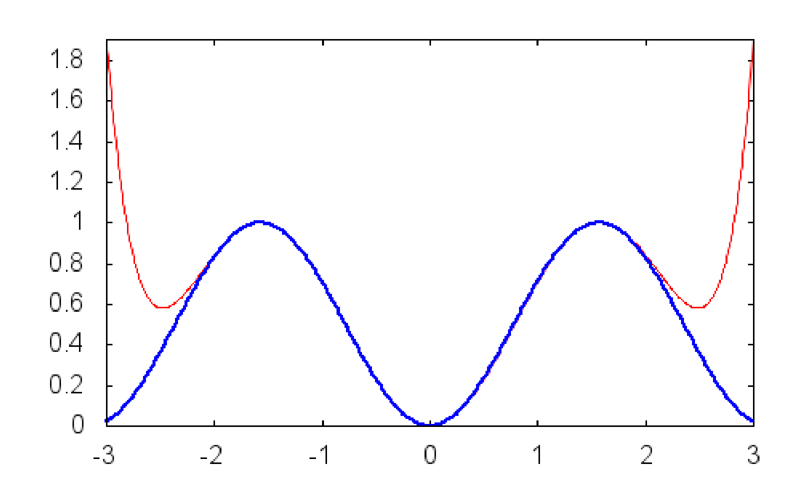


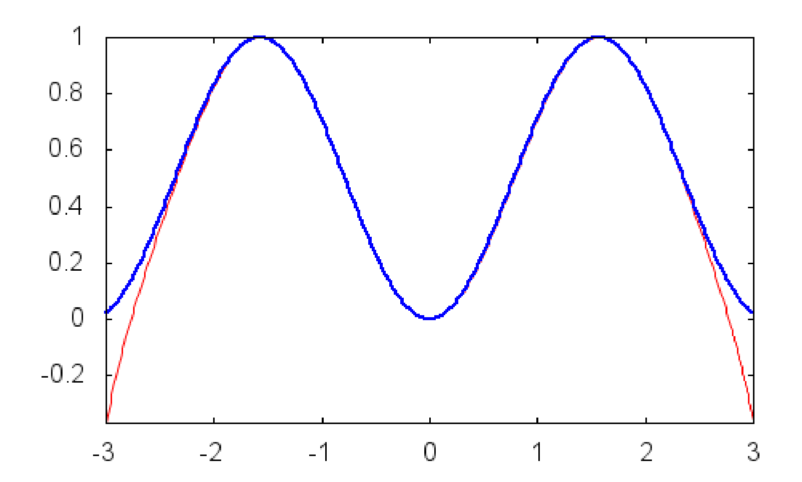


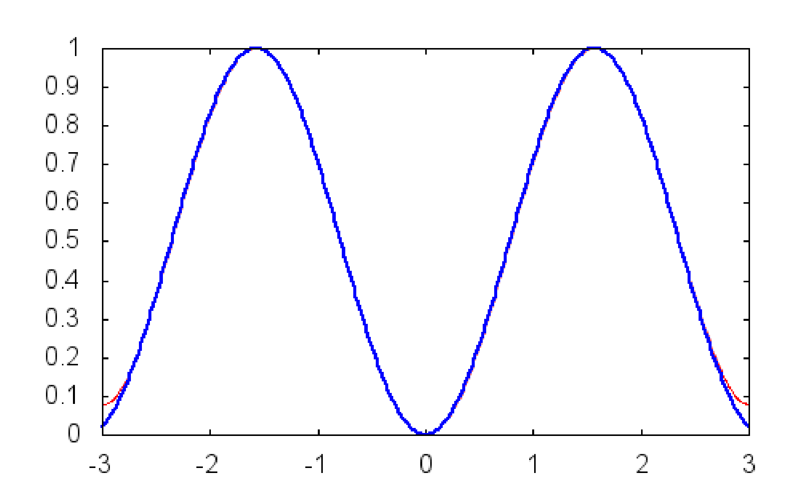

0
-2
-4
-6
-8
-10
-12
-14
-16
-18
-3
-2
-1
0
1
2
3

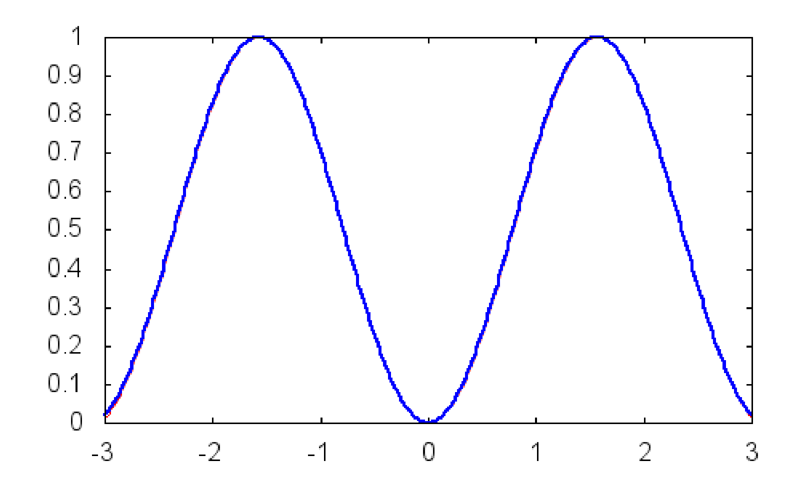

14
12
10
8
6
4
2
0
-3
-2
-1
0
1
2
3


1
0
-1
-2
-3
-4
-5
-6
-3
-2
-1
0
1
2
3


1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-3
-2
-1
0
1
2
3


1
0.8
0.6
0.4
0.2
0
-0.2
-3
-2
-1
0
1
2
3


1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
-3
-2
-1
0
1
2
3



(%o183)

Observación. Se observa como a medida que aumenta el grado de los desarrollos de Taylor estos se ajustan mejor a la función en todo el intervalo.

2.4. Ejercicios propuestos

Completar con Maxima los ejercicios siguientes:

1. Hacer una lista con los cuadrados de los 50 primeros números naturales, obtener el seno de cada uno de ellos, obtener su media aritmética y geométrica (si existe esta última).

2. Sean los vectores: $a = (2, -1, 4, 0, 3), b = (-1, 0, 2, 3, 1), c = (0, 1, -3, 4, -7)$, se pide calcular: $2a - 3b + c$, $a.b$, $a * b$, $\| a \|_1$, $\| a \|_2$ y $\| a \|_\infty$.

3. Construye una matriz 3×3, A, cuyo elemento $a_{ij} = i * j + j - i$. Extrae su segunda columna, su primera fila y el elemento $(3, 3)$. Calcula su determinante, su inversa si la tiene, su rango y su traspuesta. Obtener el núcleo y la imagen de la aplicación lineal definida por dicha matriz y sus valores y vectores propios.

4. Con vuestro número de DNI construir una matriz 3×3, A , completando con unos a la derecha hasta tener nueve cifras que repartir entre las tres filas de A. Calcular también las normas matriciales: $\| A \|_1$, $\| A \|_2$ y $\| A \|_\infty$ (la definición de estas normas se puede ver en el tercer capítulo de los apuntes).

5. Define lista1: makelist(i,i,2,21), lista2:makelist(i,i,22,31). Realiza las siguientes operaciones usando algunos de los comandos vistos.

   - Multiplica cada elemento de "lista1"por todos los elementos de "lista2". El resultado será una lista con 20 elementos (que a su vez serán listas de 10 elementos), a la que denominaréis "productos".
   - Calcula la suma de cada una de las listas que forman la lista "productos"(no te equivoques, comprueba el resultado). Obtendrás una lista con 20 números.
   - Calcula el producto de los elementos de la lista obtenida en el apartado anterior.

6. Hallar las raíces de la ecuación: $(x-1)(x-2)\dots(x-19)(x-20)+10^{-7}\cdot x^{19}=0$.

7. Averiguar si $cotg(x)=o(1/x)$ cuando x tiende a 0. Utilizando el desarrollo de McLaurin probar que e^x-1 tampoco es una $O(x^2)$ cuando x tiende a cero.

8. Dada la integral $y_n=\int_0^1 \frac{x^n}{a+x}dx$, donde a es la última cifra de vuestro DNI, en caso de ser 0 o 1 tomar $a=2$. Usar la fórmula exacta de reducción $y_n=\frac{1}{n}-ay_{n-1}$, para partiendo de $y_0=log(\frac{1+a}{a})$, calcular y_{16}, comparar con el valor exacto dado por Maxima.

9. La fórmula de reducción para la integral $I_{n+1}=\int_0^1 x^{n+1}e^x dx=e-(n+1)I_n$, permite obtener su valor partiendo de $I_0=e-1$, obtener $I_{18},I_{20},I_{21},I_{30},I_{31},I_{40},I_{41},\dots$ contradice esto el hecho de que $lim_{n\to\infty}I_n=0$. ¿Qué se puede decir de la estabilidad de este algoritmo?.

10. Dibuje con draw2d, en una misma ventana, la función $f(x)=xsin(x)^2$ y sus polinomios de Taylor de orden 2, 4, 6 y 8 todos con diferente color, siento el trazo de $f(x)$ el doble de grueso. Elija un rango adecuado para que se vea bien.

11. Dibujar las bolas de centro el origen y radio 1 para las normas usuales de $\mathbb{R}^2$.

Capítulo 3

Resolución numérica de ecuaciones no lineales

3.1. Más sobre los comandos generales de Maxima para resolver una ecuación o sistema algebraico

3.1.1. El comando "solve"

Volvamos a insistir sobre alguno de los comando generales de Maxima para resolver ecuaciones algebraicas no lineales, por ejemplo la función "**solve**" que nos resuelve una ecuación algebraica sencilla de manera exacta, bien sea lineal o no lineal. Aunque, como ya vimos, no siempre le será posible encontrar solución, como ocurre por ejemplo para las ecuaciones polinómicas de grado mayor a 4 no resolubles por radicales.

```
(%i1) solve(x^2-3*x+1=0,x);
```

(%o1) $[x = -\frac{\sqrt{5}-3}{2}, x = \frac{\sqrt{5}+3}{2}]$

También podemos resolver ecuaciones que dependan de algún parámetro, pero en este caso siempre hemos de indicar respecto de que variable resolvemos

```
(%i2) eq1:x^3-a*x^2-x^2+2*x=0;
```

(%o2) $x^3 - a\,x^2 - x^2 + 2\,x = 0$

```
(%i3) solve(eq1,x);
```

(%o3) $[x=-\frac{\sqrt{a^2+2\,a-7}-a-1}{2},x=\frac{\sqrt{a^2+2\,a-7}+a+1}{2},x=0]$

En el caso de ecuaciones de una sola variable podemos ahorrar el escribirla, pues basta poner

```
(%i4)  solve(x^2+2*x=0);
```

(%o4) $[x=-2,x=0]$

```
(%i5)  solve(x^2+2*x=0,x);
```

(%o5) $[x=-2,x=0]$

También podemos ahorrarnos el escribir el segundo término de la ecuación si éste es cero

```
(%i6)  solve(x^2+2*x);
```

(%o6) $[x=-2,x=0]$

Cuando buscamos las raíces de un polinomio a veces tenemos que tener en cuenta la multiplicidad

```
(%i7)  solve(x^7-2*x^6+2*x^5-2*x^4+x^3=0,x);
```

(%o7) $[x=-i,x=i,x=1,x=0]$
lo que se pide mediante el comando "**multiplicities**", como sigue

```
(%i8)  multiplicities;
```

(%o8) $[1,1,2,3]$

Como señalamos antes, cuando sea necesario, hay que tener cuidado de no olvidar indicarle la variable respecto de la cual resolvemos

```
(%i9)  eq1:x^3-a*x^2-x^2+2*x=0;
```

(%o9) $x^3-a\,x^2-x^2+2\,x=0$

```
(%i10) solve(eq1);
```

solve: more unknowns than equations. Unknowns given : [a,x] Equations given: $[x^3-a\,x^2-x^2+2\,x=0]$ – an error. To debug this try: debugmode(true);

```
(%i11) solve(eq1,a);
```

(%o11) $[a=\frac{x^2-x+2}{x}]$

```
(%i12) solve(eq1,x);
```

(%o12) $[x=-\frac{\sqrt{a^2+2\,a-7}-a-1}{2},x=\frac{\sqrt{a^2+2\,a-7}+a+1}{2},x=0]$

La orden solve no sólo puede resolver ecuaciones algebraicas

```
(%i13) solve(sin(x)*cos(x)=0,x);
```

$solve:using\ arc{-}trig\ functions\ to\ get\ a\ solution.\ Some\ solutions\ will\ be\ lost.$
(%o13) $[x=0,x=\frac{\pi}{2}]$

¿Qué ocurre aquí? La expresión $sen(x)\cdot cos(x)$ vale cero cuando el seno o el coseno se anulen. Para calcular la solución de sen(x)=0 aplicamos la función arcoseno a ambos lados de la ecuación. La función arcoseno vale cero en cero pero la función seno se anula en muchos más puntos. Nos estamos dejando todas esas soluciones y de eso es de lo que nos está avisando Maxima. Como cualquiera puede imaginarse, Maxima no resuelve todo. Incluso en las ecuaciones más “sencillas”, los polinomios, se presenta el primer problema: en general, no hay una fórmula en términos algebraicos para obtener las raíces de un polinomio de grado 5 o más. Pero no hay que ir tan lejos, cuando añadimos raíces, logaritmos, exponenciales, etc., la resolución de ecuaciones se complica mucho. En esas ocasiones lo más que podemos hacer es ayudar a Maxima a resolverlas.

```
(%i14) eq:x+3=sqrt(x+1);
```

(%o14) $x+3=\sqrt{x+1}$

```
(%i15) solve(eq,x);
```

(%o15) $[x=\sqrt{x+1}-3]$

```
(%i16) solve(eq^2);
```

(%o16) $[x=-\frac{\sqrt{7}\,i+5}{2},x=\frac{\sqrt{7}\,i-5}{2}]$

Uno de los ejemplos usuales en los que utilizaremos las soluciones de una ecuación es en el estudio de una función. Necesitaremos calcular puntos críticos, esto es, ceros de la derivada. El resultado de la orden solve no es una lista de puntos, es una lista de ecuaciones. Una primera solución consiste en usar la orden “**rhs**” e ir recorriendo una a una las soluciones:

```
(%i17) sol:solve(x^2-4*x+3);
```

(%o17) $[x = 3, x = 1]$

```
(%i18) rhs(part(sol,1));
```

(%o18) 3

```
(%i19) rhs(part(sol,2));
```

(%o19) 1

Este método no es práctico en cuanto tengamos un número un poco más alto de soluciones. Tenemos que encontrar una manera de aplicar la orden rhs a toda la lista de soluciones. Eso es justamente para lo que podemos utilizar la orden map:

```
(%i20) sol:map(rhs,solve(x^2-4*x+3));
```

(%o20) $[3, 1]$

También podemos resolver sistemas de ecuaciones. Sólo tenemos que escribir la lista de ecuaciones y de incógnitas. Por ejemplo:

```
(%i21) solve([x^2+y^2=1,(x-2)^2+(y-1)^2=4],[x,y]);
```

(%o21) $[[x = \frac{4}{5}, y = -\frac{3}{5}], [x = 0, y = 1]]$

Siempre hay que tener en cuenta que, por defecto, Maxima da todas las soluciones incluyendo las complejas, aunque muchas veces no pensemos en ellas. Por ejemplo, la recta $x+y = 5$ es exterior a la circunferencia $x^2+y^2 = 1$, por tanto no la corta en el plano real, pero si en el complejo.

```
(%i22) solve([x^2+y^2=1,x+y=5],[x,y]);
```

(%o22) $[[x = -\frac{\sqrt{23}\,i-5}{2}, y = \frac{\sqrt{23}\,i+5}{2}], [x = \frac{\sqrt{23}\,i+5}{2}, y = -\frac{\sqrt{23}\,i-5}{2}]]$

Observación. Para resolver sistemas lineales utilizamos preferiblemente el comando "**linsolve([ecuaciones],[incógnitas])**".

Si la solución depende de un parámetro o varios, Maxima utilizará $\%r1$, $\%r2, \cdots$ para referirse a estos. Por ejemplo:

```
(%i23) solve([x+y+z=3,x-y=z],[x,y,z]);
```

(%o23) $[[x = \frac{3}{2}, y = -\frac{2\,\%r1-3}{2}, z = \%r1]]$

¿Qué pasa si el sistema de ecuaciones no tiene solución?

```
(%i24) solve([x+y=0,x+y=1],[x,y]);
```

(%o24) []

¿Y si todos los valores de x cumplen la ecuación?

```
(%i25) solve((x+1)^2=x^2+2*x+1,x);
```

(%o25) $[x = x]$

Maxima nos dice que el sistema se reduce a $x = x$ que claramente es cierto para todo x. El siguiente caso es similar, obviamente $(x+y)^2 = x^2+2xy+y^2$, ¿qué dice al respecto Maxima?.

```
(%i26) solve((x+y)^2=x^2+2*x*y+y^2,[x,y]);
```

solve : *dependent equations eliminated* : (1) (%o26) $[[x = \%r3, y = \%r2]]$
En otras palabras, x puede tomar cualquier valor e y lo mismo.

3.1.2. El comando “algsys”

La orden “**algsys**” resuelve ecuaciones o sistemas de ecuaciones algebraicas. La primera diferencia de algsys con la orden solve es pequeña: algsys siempre tiene como entrada listas, en otras palabras, tenemos que agrupar la ecuación o ecuaciones entre corchetes igual que las incógnitas.

“**algsys([ecuaciones],[variables])**” resuelve la ecuación o ecuaciones

Si la variable opcional “**realonly**” vale **true**, algsys muestra sólo soluciones reales.

```
(%i27) eq:x^2-4*x+3;
```

(%o27) $x^2 - 4\,x + 3$

```
(%i28) algsys([eq],[x]);
```

(%o28) $[[x = 3], [x = 1]]$

La segunda diferencia es que algsys intenta resolver numéricamente la ecuación si no es capaz de encontrar la solución exacta.

```
(%i29) solve(eq:x^6+x+1);
```

(%o29) $[0 = x^6 + x + 1]$

```
(%i30) algsys([eq],[x]);
```

Salida (%o30)

$$[[x = -1{,}0383807544458461\, i - 0{,}15473514449684],$$
$$[x = 1{,}0383807544458461\, i - 0{,}15473514449684],$$
$$[x = -0{,}30050692030955\, i - 0{,}79066718881442],$$
$$[x = 0{,}30050692030955\, i - 0{,}79066718881442],$$
$$[x = 0{,}94540233331126 - 0{,}61183669378101\, i],$$
$$[x = 0{,}61183669378101\, i + 0{,}94540233331126]]$$

En general, para ecuaciones polinómicas algsys nos permite algo más de flexibilidad ya que funciona bien con polinomios de grado alto y, además, permite seleccionar las raíces reales. El comportamiento de algsys está determinado por la variable realonly. Su valor por defecto es false. Esto significa que algsys muestra todas las raíces. Si su valor es true sólo muestra las raíces reales.

```
(%i31) eq:x^4-1=0;
```

(%o31) $x^4 - 1 = 0$

```
(%i32) realonly;
```

(%o32) $false$

```
(%i33) algsys([eq],[x]);
```

(%o33) $[[x = 1], [x = -1], [x = i], [x = -i]]$

```
(%i34) realonly:true;
```

(%o34) $true$

```
(%i35) algsys([eq],[x]);
```

(%o35) $[[x = 1], [x = -1]]$

3.1.3. Los comandos "allroots" y "realroots"

Las ecuaciones polinómicas se pueden resolver de manera aproximada mediante los comandos "**allroots**" y "**realroots**", que están especializados en

encontrar soluciones racionales aproximadas de polinomios en una variable. Se escriben en la forma:

"**allroots(polinomio)**" soluciones aproximadas del polinomio

"**realroots(polinomio)**" soluciones aproximadas reales del polinomio

Estas órdenes nos dan todas las soluciones y las soluciones reales de un polinomio en una variable y pueden ser útiles en polinomios de grado alto.

```
(%i36) eq:x^9+x^7-x^4+x;
```

(%o36) $x^9 + x^7 - x^4 + x$

```
(%i37) allroots(eq);
```

Salida (%o37)

$$[x = 0{,}0, x = 0{,}30190507748312\, i + 0{,}84406777982779,$$

$$x = 0{,}84406777982779 - 0{,}30190507748312\, i,$$

$$x = 0{,}89231329168876\, i - 0{,}32846441923836,$$

$$x = -0{,}89231329168876\, i - 0{,}32846441923836,$$

$$x = 0{,}5110407920843\, i - 0{,}80986929589483,$$

$$x = -0{,}5110407920843\, i - 0{,}80986929589483,$$

$$x = 1{,}189238256723473\, i + 0{,}2942659353054,$$

$$x = 0{,}2942659353054 - 1{,}189238256723473\, i]$$

```
(%i38) realroots(eq);
```

(%o38) $[x = 0]$

3.2. A vueltas con el método de bipartición

Aunque ya vimos este método en la primera práctica, volvamos a aplicarlo para resolver, por ejemplo la ecuación $f(x) = x^6 + x - 5 = 0$ en el intervalo $[0, 2]$, utilizando el método de bipartición o bisección y determinado el número máximo de pasos a realizar. Notemos en primer lugar que $f(0) = -5$ y $f(2) = 61$, por tanto $f(x)$ posee al menos una raíz en dicho intervalo; por otro lado, como $f'(x) = 6x^5 + 1 > 0$ para todo $x \in [0, 2]$, se deduce que existe una única raíz en dicho intervalo. Seguidamente iremos subdividiendo

el intervalo por su punto medio, hasta que el valor de f en dicho punto sea muy pequeño, por ejemplo menor que el épsilon de máquina, en cuyo caso diríamos que es la "solución exacta de máquina", o bien podemos seguir hasta que el subintervalo conteniendo a la raíz tenga una amplitud menor que una magnitud *Error* dada de antemano, en cuyo caso saldríamos diciendo que la "aproximación buscada es". Supongamos que elegimos $Error = 10^{-6}$, como los extremos del intervalo son $a = 0$ y $b = 2$, sabemos que si realizamos el método n veces, la amplitud del enésimo subintervalo será en nuestro caso $b_n - a_n = (b-a)/2^n = 1/2^{n-1}$, ahora si damos como aproximación el punto medio c de este último subintervalo, el error absoluto que cometeremos será menor que $1/2^n$ y si queremos que sea menor que 10^{-6}, basta con tomar $n > log_2(10^6)$, por ejemplo $n = [log_2(10^6)] + 1$, siendo $[x]$ la parte entera del número real x. Y el programa de Maxima puede escribirse como sigue:

```
(%i39) f(x):=x^6+x-5;
```

(%o39) $f(x) := x^6 + x - 5$

```
(%i40) log2(x):=log(x)/log(2);
```

(%o40) $\log2(x) := \frac{\log(x)}{\log(2)}$

```
(%i41) a:0;
```

(%o41) 0

```
(%i42) b:2;
```

(%o42) 2

```
(%i43) Error:10^(-6);
```

(%o43) $\frac{1}{1000000}$

```
(%i44) nmaxpasos:entier(log2((b-a)/Error))+1;
```

(%o44) 21

```
(%i45) for i:1 thru nmaxpasos do
       (
       c:(a+b)/2,
       if abs(f(c))<2*10^(-16) then (print("La solucion exacta
       de máquina es "),return(c))
       else( if f(a)*f(c)<0 then b:c else a:c));
       print("La aproximación buscada es c = ",float(c))$
```

(%o45) done

La aproximación buscada es $c =$ 1,246628761291504.

El programa anterior, puede escribirse como una función mediante un block, aplicable a toda ecuación $f(x) = 0$, con f continua en $[a, b]$ y verificando $f(a)f(b)$<0, en la forma:

```
(%i48) biseccion1(f,a,b,Error):= block([numer],numer:true,
       if (sign(f(a)) = sign(f(b))) then (print("¡Atención,
       f tiene el mismo signo en ",a," y ",b,"!"), return(false)),
       log2(x):=log(x)/log(2),
       nmaxpasos:entier(log2((b-a)/Error))+1,
       for i:1 thru nmaxpasos do
           (
           c:(a+b)/2,
           if abs(f(c))<2*10^(-16) then
           (print("La solucion exacta de máquina es ",return(c)))
           else( if f(a)*f(c)<0 then b:c else a:c)
           )
       ,
       print("La aproximación buscada es c = ",c)
       )$
```

Veamos un par de ejemplos de aplicación

```
(%i49) kill(f)$ f(x):= x^6+x-5$ biseccion1(f,0,1,10^-6)$
```

¡Atención, f tiene el mismo signo en 0 y 1!

```
(%i52) f(x):= x^6+x-5$ biseccion1(f,0,2,10^-6)$
```

La aproximación buscada es $c =$ 1,246628761291504

3.2.1. El comando “find_root”

Maxima dispone de la función “**find_root(expr, x, a, b)**”, que utiliza el método de la bisección para resolver ecuaciones (aunque si la función es suficientemente suave, puede aplicar el método de regula falsi, que vemos sucintamente en el párrafo que sigue); este comando aproxima una raíz de expr o de una función f en el intervalo cerrado $[a, b]$, si la función tiene signos diferentes en los extremos del intervalo si no da un mensaje de error.

```
(%i54) find_root(x^6+x-5, x, 0, 2);
```

(%o54) 1,246628157210559

3.3. Método de Lagrange (o “regula falsi”)

El método de Lagrange, también conocido como método de las partes proporcionales o regula falsi”, consiste básicamente en reemplazar la gráfica de f restringida al intervalo $[a, b]$ por la recta pasando por los puntos extremos $A(a, f(a))$ y $B(b, f(b))$. Es decir se sustituye la función f por un polinomio de grado uno $p(x)$ y se resuelve la ecuación $p(x) = 0$; el punto de intersección de la recta AB con el eje de las x será la primera aproximación x_1 de la raíz buscada; luego se determina en cual de los subintervalos $[a, x_1]$ o $[x_1, b]$ está la raíz, y a este se le vuelve a aplicar el mismo método para obtener la segunda aproximación x_2, y así sucesivamente hasta obtener una aproximación que consideremos adecuada. Aplicaremos este método para obtener la solución aproximada de la ecuación $6x - e^x = 0$ en el intervalo $[2, 3]$.

```
(%i55) a1:2$ b1:3$ f(x):=6*x-exp(x);
```

(%o57) $\mathrm{f}(x) := 6\,x - \exp(x)$

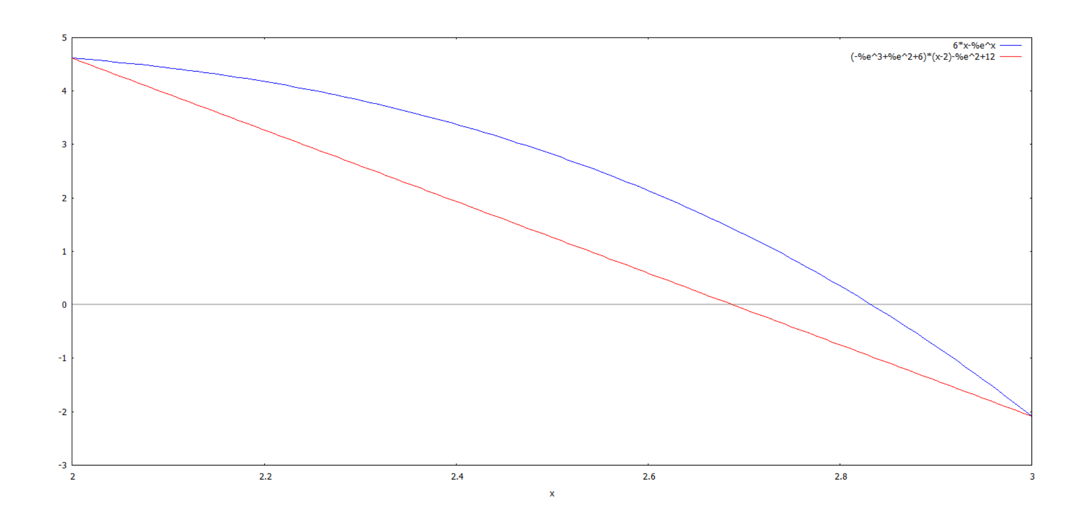
Calculemos y representemos la recta (en rojo) que une los puntos extremos $(a_1, f(a_1)) - (b_1, f(b_1))$ frente a la función $f(x)$ (en azul) en el intervalo $[a_1, b_1]$

```
(%i58) r1(x):=f(a1)+(x-a1)*(f(b1)-f(a1))/(b1-a1);
```

(%o58) $\mathrm{r1}(x) := \mathrm{f}(a1) + \frac{(x - a1)\,(\mathrm{f}(b1) - \mathrm{f}(a1))}{b1 - a1}$

```
(%i59) plot2d([f(x),r1(x)],[x,a1,b1]);
```

(%o59)



Por la gráfica se ve que la solución de $f(x) = 0$ está en el intervalo $[p1, b1]$, siendo $p1$ el punto de corte de $r1(x)$ con el eje de abscisas. Al ser el extremo izquierdo del intervalo donde está la solución, lo llamamos $a2$.

```
(%i60) a2:(a1*f(b1)-b1*f(a1))/(f(b1)-f(a1)),numer;
```

(%o60) 2,688562249647141

```
(%i61) b2:b1;
```

(%o61) 3

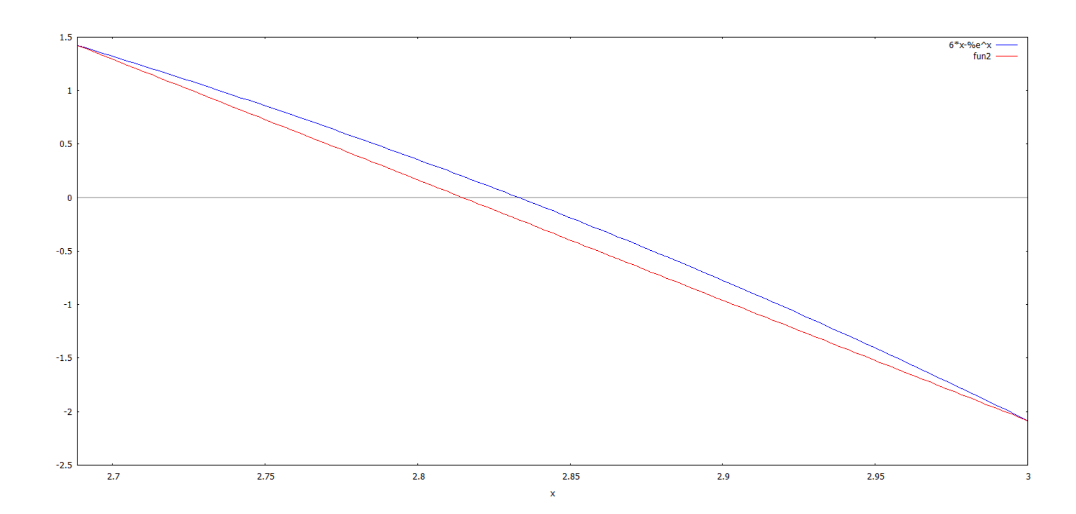
Ahora repetimos el proceso de antes para el nuevo intervalo

```
(%i62) r2(x):=f(a2)+(x-a2)*(f(b2)-f(a2))/(b2-a2);
```

(%o62) $\mathrm{r2}\,(x) := \mathrm{f}\,(a2) + \dfrac{(x - a2)\;(\mathrm{f}\,(b2) - \mathrm{f}\,(a2))}{b2 - a2}$

```
(%i63) plot2d([f(x),r2(x)],[x,a2,b2]);
```

(%o63)



Otra vez el intervalo es $[p2, b1]$ además se observa que ahora las dos gráficas están más próximas. Seguimos el proceso hasta que se vea que las dos gráficas cortan al eje OX prácticamente en el mismo punto (que es la solución de $f(x) = 0$ buscada).

3.4. Método de Newton-Raphson

3.4.1. Método de Newton-Raphson en una variable

Sea la ecuación $e^{-x} - x = 0$, queremos aproximar la solución de dicha ecuación en el intervalo $[0, 1]$ por el método de Newton. En primer lugar, veamos que podemos aplicar el teorema de convergencia global-2 de dicho método y por tanto que podemos comenzar a iterar por cualquier punto del intervalo dado, por ejemplo por el punto $x0 = 0{,}5$. Para ello, comencemos escribiendo la función y calculando sus derivadas primera y segunda

```
(%i64) f(x):=%e^(-x)-x;
```

(%o64) $\mathrm{f}(x) := e^{-x} - x$

```
(%i65) define(df(x),diff(f(x),x));
       define(d2f(x),diff(f(x),x,2));
       df(0);df(1.0);
```

(%o65) df $(x) := -e^{-x} - 1$
(%o66) d2f $(x) := e^{-x}$
(%o67) -2
(%o68) $-1{,}367879441171442$

Puesto que la derivada segunda es positiva en todo el intervalo la primera es estrictamente creciente, ahora bien como $f'(1) = df(1) < 0$ se sigue que la derivada primera es menor que 0 en todo el intervalo $[0, 1]$, por tanto $f(x) = 0$ posee una única raíz en dicho intervalo. Además, como $max\{|f(0)/df(0)|, |f(1)/df(1)|\} \leq \frac{1}{2} < 1$, el método de Newton converge partiendo de cualquier valor inicial en $[0, 1]$, tomemos pues $x0 = 0{,}5$.

```
(%i69) max(abs(f(0)/df(0)),abs(f(1)/df(1)));
```

(%o69) $\frac{1}{2}$

La derivada de la función en $x0$ y la recta tangente en dicho punto serán

```
(%i70) x0:0.5; df(x0);
```

(%o70) 0,5
(%o71) -1.606530659712633

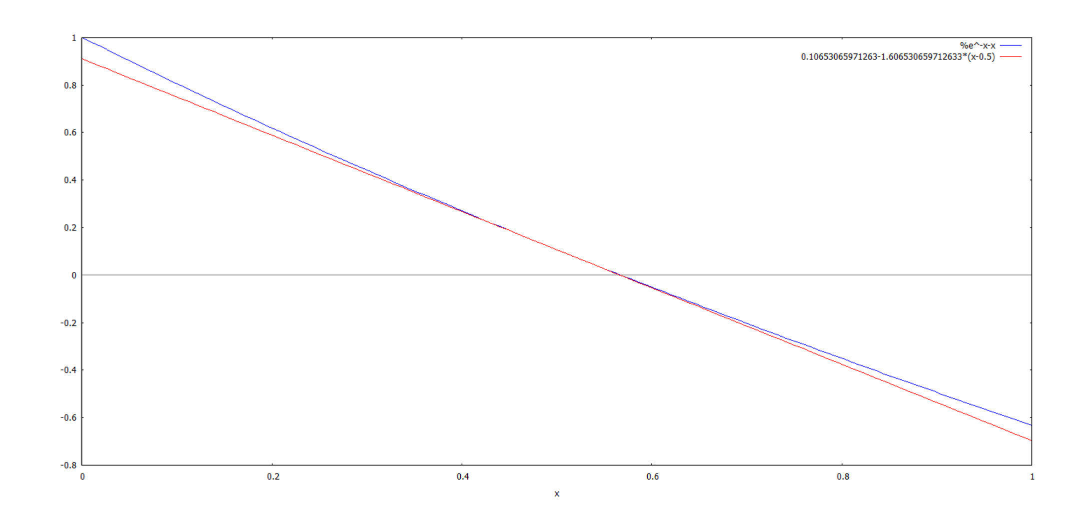
```
(%72)  r0(x):=f(x0)+df(x0)*(x-x0);
```

(%o72) $r0(x) := f(x0) + df(x0)(x - x0)$

En tanto que las gráficas que siguen, de la función (en color azul) y de la recta tangente (en rojo) en el intervalo $[0, 1]$, ponen de manifiesto la extraordinaria aproximación del método en este caso.

(%i73) `plot2d([f(x),r0(x)],[x,0,1]);`

(%o73)



El cálculo de la abscisa del punto de corte de la tangente con el eje OX viene dado por

(%i74) `x1:x0-f(x0)/df(x0);`

(%o74) 0,56631100319722

Realizamos la segunda iteración, para ello calculamos en primer lugar la pendiente en $x1$ y la recta tangente en el punto correspondiente a esta nueva abscisa

(%i75) `df(x1);`

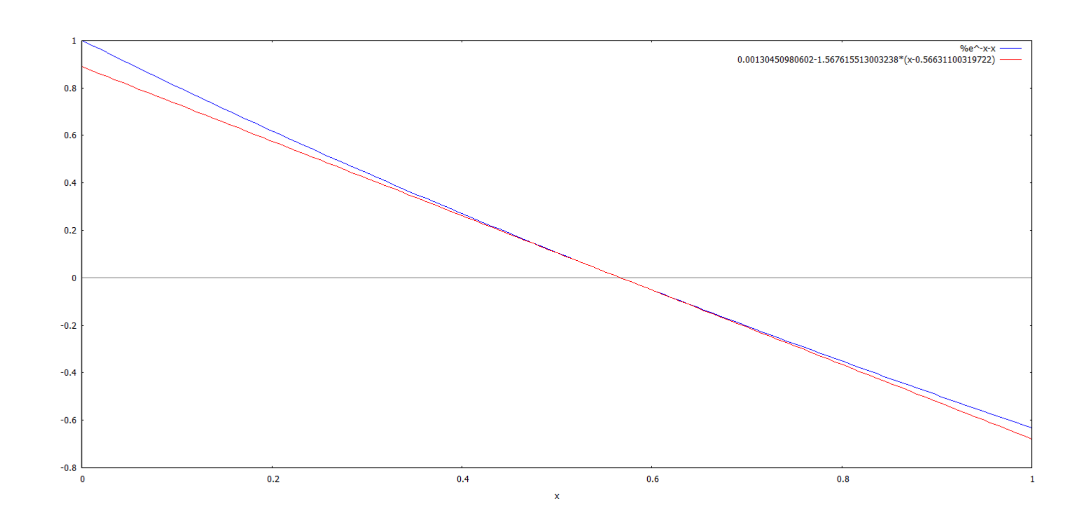
(%o75) $-1{,}567615513003238$

(%i76) `r1(x):=f(x1)+df(x1)*(x-x1);`

(%o76) $r1(x) := f(x1) + df(x1)(x - x1)$

(%i77) `plot2d([f(x),r1(x)],[x,0,1]);`

(%o77)



Estando dada la segunda aproximación de la raíz buscada por

```
(%i78) x2:x1-f(x1)/df(x1);
```

(%o78) 0,56714316503486

Para la tercera iteración procedemos de la misma manera, omitiendo ahora la gráfica, obteniendo

```
(%i79) df(x2);
```

(%o79) $-1{,}567143361515334$

```
(%i80) r2(x):=f(x2)+df(x2)*(x-x2);
```

(%o80) $r2(x) := f(x2) + df(x2)(x - x2)$

```
(%i81) x3:x2-f(x2)/df(x2);
```

(%o81) 0,56714329040978

Finalmente, podemos acotar el error de $x3$ en la forma $|x3-r| \leq |f(x3)|/m$, siendo $0 < m \leq |f'(x)|$ para todo $x \in [0,1]$ y r la raíz exacta buscada.

```
(%i82) print("El error absoluto de x3 <= ",abs(f(x3)/1.3))$
```

El error absoluto de $x3 \leq 3{,}41607084500048\,14\,10^{-15}$

3.4.2. Funciones para el método de Newton-Raphson

Supongamos que se cumplen condiciones suficientes para aplicar el método de Newton para aproximar la raíz de una ecuación $f(x) = 0$ en un intervalo $[a, b]$, partiendo de un punto $x0$ (lo cual se puede estudiar a priori, de acuerdo con los resultados vistos en teoría), vamos a escribir una función que nos dé una aproximación de la raíz parando el programa cuando se sobrepase el número máximo de iteraciones previstas o bien cuando la diferencia entre dos aproximaciones consecutivas sea menor que un cierto épsilon dado, podríamos hacerlo mediante el siguiente block.

```
(%i83) newton0(f,x0,epsilon,nmax):= block(
       [numer],numer:true,x[0]:x0,
       define(Df(x),diff(f(x),x)),
       for i:1 thru nmax do
       (
       x[i]:x[i-1]-f(x[i-1])/Df(x[i-1]),
       if abs(x[i]-x[i-1])< epsilon then
       return((print("La aproximación buscada es
       x(",i,") = ",x[i])))
       ),
       if abs(x[nmax]-x[nmax-1]) >= epsilon then
       print("No lograda la aproximación deseada en ",
       nmax," iteraciones")
       )$
```

```
(%i84) f(x):=%e^-x-x;newton0(f,0.5,10^-8,20)$
```

(%o84) $f(x) = e^{-x} - x$
La aproximación buscada es $x(4) = 0{,}56714329040978$

En tanto que el error absoluto de esta aproximación será menor que 10^{-16}, como se ve en los cálculos que siguen.

```
(%i86) abs(f(x[4]))/1.3;
```

(%o86) $8{,}5401771125012034\,10^{-17}$

Si ponemos $nmax = 3$, vemos que se detiene el programa por haber superado el número máximo de iteraciones sin conseguir la aproximación deseada

```
(%i87) f(x):=%e^-x-x; newton0(f,0.5,10^-8,3)$
```

(%o87) $f(x) := e^{-x} - x$

No lograda la aproximación deseada en 3 iteraciones

Observación. El programa anterior puede utilizarse para resolver aproximadamente ecuaciones no lineales $f(x) = 0$, por el método de Newton, si no se obtiene la aproximación, podríamos intentar cambiar el número máximo de pasos (nmax) o el punto inicial ($x0$).

La función newton de Maxima. Hay también una función de Maxima para el método de Newton, se trata de la siguiente: "**newton(expresion, x, x0, eps)**", pero requiere ejecutar previamente "**load(newton1)**" para cargar el paquete correspondiente. El resultado es una raíz aproximada de expresion, en la variable x, comienza a iterar por $x0$ y el proceso sigue hasta que se cumple que $|expresion| < eps$. Para información más detallada sobre este comando teclear ? newton. Veamos su aplicación al ejemplo anterior.

```
(%i88) load(newton1);newton(%e^-x-x,x,0.5,10^-8);
```

(%o88) C:/PROGRA 2/MAXIMA 1.0-2/share/maxima/5.28.0-2/share/numeric/newton1.mac
(%o89) 0.56714329040978

3.4.3. Método de Newton-Raphson en varias variables

Dado el sistema

$$\begin{aligned} f_1(x_1, x_2) = 0 \\ f_2(x_1, x_2) = 0 \end{aligned}$$

con $f(x) = f(x_1, x_2) = (f_1(x_1, x_2), f_2(x_1, x_2))$ de clase $\mathbb{C}^{(3)}$ en un entorno de la raíz buscada $r = (r_1, r_2)$ y con jacobiano inversible en r, pasamos al sistema equivalente

$$\begin{aligned} x_1 = g_1(x_1, x_2) \\ x_2 = g_2(x_1, x_2) \end{aligned}$$

donde ahora es $g(x) = x - J_f^{-1}(x)f(x)$. El método definido por esta g se denomina método de **Newton para sistemas**, es localmente convergente con convergencia de segundo orden y viene dado por el algoritmo

$$\boxed{x^{(m+1)} = x^{(m)} - J_f^{-1}(x^{(m)})f(x^{(m)})}$$

La aplicación de este método requiere que $J_f(x^{(m)})$ sea inversible para todo m y en la práctica se suele presentar en la forma

$$J_f(x^{(m)})(x^{(m+1)} - x^{(m)}) = -f(x^{(m)})$$

y llamando $\delta^{(m)} = x^{(m+1)} - x^{(m)}$, el método consiste en

$$\begin{array}{c}\text{Hallar } \delta^{(m)} \text{ verificando el sistema lineal} \\ J_f(x^{(m)})\delta^{(m)} = -f(x^{(m)}) \\ \text{y obtener la aproximación siguiente en la forma} \\ x^{(m+1)} = x^{(m)} + \delta^{(m)} \ (m \geq 0)\end{array}$$

A modo de ejemplo, vamos a aproximar la solución del sistema: $x^2 + y^2 - 2 = 0$, $x^2 - y + 1 = 0$ en el primer cuadrante, utilizando el método de Newton-Raphson comenzando a iterar por el punto $(x0, y0) = (0{,}5, 1)$ y haciendo tres iteraciones. Ahora, identificaremos $x_1 \equiv x$, $x_2 \equiv y$, $\delta_1 \equiv d_1$ y $\delta_2 \equiv d_2$ y calculamos la matriz jacobiana

```
(%i90) kill(all)$
       f:matrix([x^2+y^2-2],[x^2-y+1]);
       d:matrix([d1],[d2]);
       J:jacobian([x^2+y^2-2,x^2-y+1],[x,y]);
       J.d=-f;
```

(%o1) $\begin{pmatrix} y^2 + x^2 - 2 \\ -y + x^2 + 1 \end{pmatrix}$

(%o2) $\begin{pmatrix} d1 \\ d2 \end{pmatrix}$

(%o3) $\begin{pmatrix} 2\,x & 2\,y \\ 2\,x & -1 \end{pmatrix}$

(%o4) $\begin{pmatrix} 2\,d2\,y + 2\,d1\,x \\ 2\,d1\,x - d2 \end{pmatrix} = \begin{pmatrix} -y^2 - x^2 + 2 \\ y - x^2 - 1 \end{pmatrix}$

Seguidamente le damos valores iniciales a x e y y calculamos los primeros incrementos

```
(%i5)  numer:true$ x:0.5$ y:1$
       linsolve([2*x*d1+2*y*d2=-x^2-y^2+2,
       2*x*d1-d2=-x^2+y-1],[d1,d2]);
```

(%o8) $[d1 = 0{,}083333333333333, d2 = 0{,}33333333333333]$

Hallamos la primera aproximación en la forma

```
(%i9)  x:x+0.083333333333333;y:y+0.33333333333333;
```

(%o9) 0,58333333333333
(%o10) 1,33333333333333

Ahora realizamos la segunda iteración, partiendo de los valores actualizados obtenemos los nuevos incrementos

```
(%i11) linsolve([2*x*d1+2*y*d2=-x^2-y^2+2,
       2*x*d1-d2=-x^2+y-1],[d1,d2]);
```

(%o11) $[d1 = -0{,}031926406926407, d2 = -0{,}030303030303027]$

con lo que las nuevas aproximaciones serán

```
(%i12) x:x-0.031926406926407;y:y-0.030303030303027;
```

(%o12) 0,55140692640693
(%o13) 1,303030303030303

Repitiendo para una tercera iteración tendremos los incrementos

```
(%i14) linsolve([2*x*d1+2*y*d2=-x^2-y^2+2,
       2*x*d1-d2=-x^2+y-1],[d1,d2]);
```

(%o14) $[d1 = -0{,}0011551748007612, d2 = -2{,}5464731347085587\, 10^{-4}]$

Resultando la tercera iteración que sigue

```
(%i15) x:x-0.0011551748007612;y:y-2.5464731347085587*10^-4;
```

(%o15) 0,55025175160616
(%o16) 1.302775655716832

Podemos comprobar en que medida verifica las ecuaciones la tercera aproximación obtenida

```
(%i17) f:matrix([x^2+y^2-2],[x^2-y+1]);
```

(%o17) $\begin{pmatrix} 1{,}3992740743873355\, 10^{-6} \\ 1{,}3344288203320787\, 10^{-6} \end{pmatrix}$

Finalmente, lo hacemos directamente y lo comparamos con la solución del primer cuadrante, que es una buena aproximación de la segunda de las que siguen:

```
(%i18) kill(all)$algsys([x^2+y^2-2=0,x^2-y+1=0],[x,y]);
```

(%o1) $[[x = -0{,}55025052270034, y = 1{,}302775637731995],$
$[x = 0{,}55025052270034, y = 1{,}302775637731995],$
$[x = -1{,}817354021023971\, i, y = -2{,}302775637731995],$
$[x = 1{,}817354021023971\, i, y = -2{,}302775637731995]]$

Función newton de Maxima para sistemas. Hay también una función de Maxima del método de Newton para sistemas no lineales, se trata de la siguiente: "**newton ([ecuaciones],[variables],[aproximacionesiniciales])**", donde [ecuaciones] es la lista de ecuaciones a resolver, [variables] es la lista con los nombres de las incógnitas y [aproximacionesiniciales] es la lista de aproximaciones iniciales, también se requiere ejecutar previamente "**load("mnewton")**" para cargar el paquete correspondiente. La solución se devuelve en el mismo formato que lo hace la función "solve()". Si no se encuentra solución al sistema, se obtiene "[]" como respuesta. Para una información más completa sobre este comando teclear ? mnewton. Veamos su aplicación al ejemplo anterior.

```
(%i2)  load("mnewton");mnewton([x^2+y^2-2,x^2-y+1],[x,y],[0.5,1]);
```

(%o2) C:/PROGRA 2/MAXIMA 1.0-2/share/maxima/5.28.0-2/share/mnewton/mnewton.mac

(%o3) $[[x = 0{,}55025052270034, y = 1{,}302775637731995]]$

3.5. Ejercicios propuestos

Completar con Maxima los ejercicios siguientes:

1. Utilizar alguno de los comandos de Maxima para:

   a) Resolver la ecuación no lineal $x^4 + 7x^3 + 2x + 1 = 0$.

   b) Hallar las raíces reales del polinomio $s(x) = x^7 - x^6 + 5x^2 + x + 2$.

   c) Obtener todas las raíces del polinomio $x^8 - 36x^7 + 546x^6 - 4536x^5 + 22449x^4 - 67284x^3 + 118124x^2 - 109584x + 40320$ y también las del polinomio perturbado: $x^8 - 36{,}001x^7 + 546x^6 - 4536x^5 + 22449x^4 - 67284x^3 + 118124x^2 - 109584x + 40320$.

   d) Resolver el sistema de ecuaciones

$$\begin{cases} x^2 + y^2 = 9, \\ y = 3x^2. \end{cases}$$

2. Aplicar el método de bisección para hallar una raíz aproximada de la ecuación $6x - e^x = 0$ en el intervalo $[2, 3]$ con un error menor que 10^{-7}.

3. Realizar 3 iteraciones del método de Regula Falsi para hallar una raíz de la ecuación $x^6 + x - 5 = 0$ en el intervalo $[0, 2]$.

4. Resolver la ecuación $x^3 - 3x - 2 = 0$ por el método de Newton empezando por el punto 3 para calcular la raíz positiva $r = 2$ del polinomio. Verificar que el orden numérico de convergencia es $p = 2$.

5. Para la ecuación $x^3 - 3x - 2 = 0$, utilizar el método de Newton empezando por el punto -2 para calcular la raíz doble negativa $r = -1$ del polinomio. Verificar que el orden numérico de convergencia, en este caso de raíz múltiple, se convierte en lineal $p = 1$.

6. Resolver la ecuación $x^3 - 3x - 2 = 0$ por el método de la secante empezando por los puntos $x_0 = 4$ y $x_1 = 3$ para calcular la raíz positiva $r = 2$ del polinomio. Verificar que el orden numérico de convergencia es $p = \frac{1+\sqrt{5}}{2} \approx 1{,}6$.

7. Para la ecuación $x^3 - 3x - 2 = 0$, utilizar el método de la secante empezando por los puntos -2 y $-1{,}5$ para aproximar la raíz doble negativa $r = -1$ del polinomio. Verificar que el orden numérico de convergencia, en este caso de raíz múltiple, se convierte en lineal $p = 1$.

8. Se tiene una función de iteración $g : I \to I$ que define un método de punto fijo convergente $x_{n+1} = g(x_n)$ que resuelve el problema $f(x) = 0$. Suponer que la función de iteración g es contractiva ya que cumple $|g'(x)| \leq k = 0{,}7 < 1$. Es decir, la constante de contractividad es k. Como el método es convergente se sabe que la distancia del iterado x_n a la raíz r puede estimarse por

$$|x_n - r| \leq \frac{k}{1-k}|x_n - x_{n-1}| \leq \frac{k^n}{1-k}|x_1 - x_0|.$$

Suponer ahora que $I = [0, 1]$, que tomamos $x_0 = 0$ y que $g(x_0) = 0{,}5$. Determinar usando wxmaxima cuántas iteraciones son necesarias para asegurar un error menor que 10^{-5}.

Capítulo 4

Resolución numérica de sistemas lineales

4.1. Introducción

Como ya vimos, la resolución de sistemas lineales, se puede hacer con solve, como puede verse en el ejemplo que sigue.

```
(%i1) solve([x-y=3,x+y=1],[x,y]);
```

(%o1) $[[x = 2, y = -1]]$

Pero se dispone de un comando más eficiente, que funciona como solve, se trata del comando "**linsolve([ecuaciones],[variables])**" que resuelve el sistema lineal[ecuaciones] respecto a las variables[variables], como se ve en el ejemplo:

```
(%i2) linsolve([x-y=3,x+y=1],[x,y]);
```

(%o2) $[x = 2, y = -1]$

También se puede definir el sistema de ecuaciones separadamente.

```
(%i3) p:[3*x+5*y-4*z=1,x+2*y+3*z=-2,-4*x+y-3*z=4];
```

(%o3) $[-4\,z + 5\,y + 3\,x = 1, 3\,z + 2\,y + x = -2, -3\,z + y - 4\,x = 4]$

```
(%i4) linsolve(p,[x,y,z]);
```

(%o4) $[x = -\frac{61}{108}, y = \frac{11}{108}, z = -\frac{59}{108}]$

No hay problema cuando hay infinitas soluciones.

```
(%i5) p:[x+y+3*z=1, 3*x+5*y-z=2, -x-3*y+7*z=0];
```

(%o5) $[3\,z+y+x=1,-z+5\,y+3\,x=2,7\,z-3\,y-x=0]$

```
(%i6) linsolve(p,[x,y,z]);
```

solve: dependent equations eliminated: (1)

(%o6) $[x=-\frac{16\,\%r1-3}{2},y=\frac{10\,\%r1-1}{2},z=\%r1]$

En este caso sólo 2 ecuaciones son linealmente independientes y habrá infinitas soluciones. Máxima llama %r1 al parámetro arbitrario.

Cuando no hay soluciones, Maxima lo indica como sigue:

```
(%i7) p:[x+y=3,2*x+2*y=5];
```

(%o7) $[y+x=3,2\,y+2\,x=5]$

```
(%i8) linsolve(p,[x,y]);
```

(%o8) []

```
(%i9) coefmatrix(p,[x,y]);
```

(%o9) $\begin{pmatrix}1 & 1\\ 2 & 2\end{pmatrix}$

Veamos un poco de álgebra matricial, para aprender a introducir matrices con las que después trabajar

```
(%i10) A:matrix([1,0,-2], [2,3,2], [-2,0,1]);
```

(%o10) $\begin{pmatrix}1 & 0 & -2\\ 2 & 3 & 2\\ -2 & 0 & 1\end{pmatrix}$

Para definir una matriz, wxMaxima nos ofrece una interfaz muy atractiva, a la que se puede acceder a través del menú Algebra!Introducir matriz... wxMaxima nos preguntará el número de filas y columnas, así como el formato de la matriz resultante (de tipo general, diagonal, simétrico o antisimétrico) y a continuación nos ofrecerá una ventana en la que podremos introducir sus elementos.Para acceder a los elementos de una matriz, podemos utilizar los corchetes.

```
(%i11) A[1,3];
```

(%o11) -2

Existe una segunda forma de definir matrices cuyos elementos se ajusten a una regla predefinida. Para ello debemos predefinir esta regla, de forma que defina una "tabla de valores", que asocie a cada fila y columna (entre corchetes) un número real, como en el siguiente ejemplo:

```
(%i12) T[i,j]:=i+j-1;
```

(%o12) $T_{i,j} := i + j - 1$

Obsérvese que, en la definición de esta tabla, hemos utilizado el operador (:=). A continuación, podemos usar la función genmatrix, a la que se puede acceder a través de Algebra!Generar matriz... en wxMaxima. Esta función nos pedirá el nombre de la tabla de valores, así como el número de filas y columnas de la matriz resultante, por ejemplo:

```
(%i13) B:genmatrix(T,3,5);
```

(%o13) $$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \end{pmatrix}$$

Utilizando submatrix se obtienen submatrices, para lo cual introducimos las filas a eliminar (si existe alguna), a continuación el nombre de la matriz y por último las columnas a eliminar (en su caso). Las funciones row y col nos permiten acceder, respectivamente, a filas y columnas que deseemos. Se pueden añadir filas y columnas nuevas con addrow y addcol, todo lo cual muestran los siguientes ejemplos:

```
(%i14) submatrix(1,B,4,5);
```

(%o14) $$\begin{pmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{pmatrix}$$

```
(%i15) row(B,1);
```

(%o15) $$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

```
(%i16) col(B,5);
```

(%o16) $$\begin{pmatrix} 5 \\ 6 \\ 7 \end{pmatrix}$$

```
(%i17) addcol(B,[10,11,12]);
```

(%o17) $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 10 \\ 2 & 3 & 4 & 5 & 6 & 11 \\ 3 & 4 & 5 & 6 & 7 & 12 \end{pmatrix}$

Veamos algunas funciones útiles:

```
(%i18) transpose(A);
```

(%o18) $\begin{pmatrix} 1 & 2 & -2 \\ 0 & 3 & 0 \\ -2 & 2 & 1 \end{pmatrix}$

```
(%i19) ident(3);
```

(%o19) $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

```
(%i20) identfor(A);
```

(%o20) $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

```
(%i21) matrix_size(A);
```

(%o21) $[3, 3]$

```
(%i22) zeromatrix(3,5);
```

(%o22) $\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$

```
(%i23) zerofor(A);
```

(%o23) $\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$

```
(%i24) mat_trace(A);
```

(%o24) 5

```
(%i25) diag_matrix(1,2,3);
```

(%o25) $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}$

```
(%i26) diagmatrix(4,5);
```

(%o26) $\begin{pmatrix} 5 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 5 \end{pmatrix}$

Puesto que Maxima es un programa de cálculo simbólico, las matrices no se limitan a almacenar valores numéricos, sino que estos pueden venir dados por cualquier tipo de expresión:

```
(%i27) C:matrix([x^2, 1+x/y], [sqrt(x), x^2-y=0]);
```

(%o27) $\begin{pmatrix} x^2 & \frac{x}{y}+1 \\ \sqrt{x} & x^2-y=0 \end{pmatrix}$

```
(%i28) C,x=2,y=4;
```

(%o28) $\begin{pmatrix} 4 & \frac{3}{2} \\ \sqrt{2} & 0=0 \end{pmatrix}$

```
(%i29) solve(C[2,2],x);
```

(%o29) $[x=-\sqrt{y}, x=\sqrt{y}]$

Las operaciones algebraicas habituales están también disponibles para matrices, aunque debemos tener en cuenta que: El operador asterisco (*) se interpreta como producto elemento a elemento. El producto matricial viene dado a través del operador punto (.). El operador ^ se utiliza para calcular las potencias de los elementos de una matriz. El operador ^^ se emplea para calcular potencias de matrices.

```
(%i30) A:matrix([a,b,c],[d,e,f]);
```

(%o30) $\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$

```
(%i31) B:matrix([u,v,w],[x,y,z]);
```

(%o31) $\begin{pmatrix} u & v & w \\ x & y & z \end{pmatrix}$

```
(%i32) A+B;
```

(%o32) $\begin{pmatrix} u+a & v+b & w+c \\ x+d & y+e & z+f \end{pmatrix}$

```
(%i33) 2*A;
```

(%o33) $\begin{pmatrix} 2\,a & 2\,b & 2\,c \\ 2\,d & 2\,e & 2\,f \end{pmatrix}$

```
(%i34) A*B;
```

(%o34) $\begin{pmatrix} a\,u & b\,v & c\,w \\ d\,x & e\,y & f\,z \end{pmatrix}$

```
(%i35) C:submatrix(B,3);
```

(%o35) $\begin{pmatrix} u & v \\ x & y \end{pmatrix}$

```
(%i36) C.A;
```

(%o36) $\begin{pmatrix} d\,v+a\,u & e\,v+b\,u & f\,v+c\,u \\ d\,y+a\,x & e\,y+b\,x & f\,y+c\,x \end{pmatrix}$

```
(%i37) A^n;
```

(%o37) $\begin{pmatrix} a^n & b^n & c^n \\ d^n & e^n & f^n \end{pmatrix}$

```
(%i38) %,n=1/2;
```

(%o38) $\begin{pmatrix} \sqrt{a} & \sqrt{b} & \sqrt{c} \\ \sqrt{d} & \sqrt{e} & \sqrt{f} \end{pmatrix}$

```
(%i39) C^^2;
```

(%o39) $\begin{pmatrix} v\,x+u^2 & v\,y+u\,v \\ x\,y+u\,x & y^2+v\,x \end{pmatrix}$

Otras funciones aplicables sobre matrices:

- **diagmatrix** y **zeromatrix**, se pueden utilizar para construir, respectivamente matrices diagonales (con todos sus elementos diagonales iguales entre sí) y matrices nulas.
- **transpose**, devuelve la matriz traspuesta (disponible en wxMaxima a través de Algebra!Trasponer matriz)
- **determinant**, calcula el determinante de una matriz cuadrada (Algebra ! Determinante)
- **rank**, calcula el rango
- **invert**, devuelve la matriz inversa (menú Algebra!Invertir matriz)
- **triangularize**, devuelve una matriz triangular superior resultante de aplicar el método de Gauss a una matriz dada

- **eigenvalues**, devuelve dos listas, la primera formada por los autovalores de una matriz y la segunda por sus multiplicidades (accesible desde wxMaxima en Algebra!Valores propios)
- **eigenvectors**, devuelve una lista formada por los autovalores junto a una serie de listas representando a sus autovectores asociados (menú Algebra ! Vectores propios de wxMaxima)

Algunos ejemplos:

```
(%i40) A:matrix([1,0,-2], [2,3,2], [-2,0,1]);
```

(%o40) $\begin{pmatrix} 1 & 0 & -2 \\ 2 & 3 & 2 \\ -2 & 0 & 1 \end{pmatrix}$

```
(%i41) determinant(A);
```

(%o41) -9

```
(%i42) B:invert(A);
```

(%o42) $\begin{pmatrix} -\frac{1}{3} & 0 & -\frac{2}{3} \\ \frac{2}{3} & \frac{1}{3} & \frac{2}{3} \\ -\frac{2}{3} & 0 & -\frac{1}{3} \end{pmatrix}$

```
(%i43) I:A.B;
```

(%o43) $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

```
(%i44) M:A-x*I;
```

(%o44) $\begin{pmatrix} 1-x & 0 & -2 \\ 2 & 3-x & 2 \\ -2 & 0 & 1-x \end{pmatrix}$

```
(%i45) solve(determinant(M)=0) /* Autovalores */;
```

(%o45) $[x=3, x=-1]$

```
(%i46) M,x=3;
```

(%o46) $\begin{pmatrix} -2 & 0 & -2 \\ 2 & 0 & 2 \\ -2 & 0 & -2 \end{pmatrix}$

```
(%i47) rank(%);
```

(%o47) 1

```
(%i48) eigenvalues(A) /* [autovalores], [multiplicidades] */;
```

(%o48) $[[3,-1],[2,1]]$

```
(%i49) eigenvectors(A) /*[autovalores], [v1], [v2], [v3]*/;
```

(%o49) $[[[3,-1],[2,1]],[[[1,0,-1],[0,1,0]],[[1,-1,1]]]]$

```
(%i50) triangularize(A);
```

(%o50) $$\begin{pmatrix} 1 & 0 & -2 \\ 0 & 3 & 6 \\ 0 & 0 & -9 \end{pmatrix}$$

El comando eigenvalues llama a solve y por tanto, a veces, no devuelve resultado, veámoslo con el siguiente ejemplo

```
(%i51) R[i,j]:= j^(i-1);A:genmatrix(R,5,5);
```

(%o51) $R_{i,j} := j^{i-1}$

(%o52) $$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 4 & 9 & 16 & 25 \\ 1 & 8 & 27 & 64 & 125 \\ 1 & 16 & 81 & 256 & 625 \end{pmatrix}$$

```
(%i53) eigenvalues(%);
```

solve is unable to find the roots ofthe characteristic polynomial.

(%o53) []

Una alternativa, es hacer su polinomio característico y resolverlo por ejemplo con algsys

```
(%i54) algsys([charpoly(A,x)=0],[x]);
```

(%o54) $[[x=680{,}9266055045872],[x=17{,}65848670756646],[x=1{,}973060344827586],[x=0{,}41235602094241],[x=0{,}029439197854728]]$

4.2. Métodos de factorización. Normas matriciales.

Habida cuenta de la sencillez y eficiencia computacional de la resolución de sistemas lineales cuya matriz de coeficientes sea triangular, tanto con matriz triangular inferior como superior (en cualquier caso conlleva un total de n^2 operaciones la resolución de uno de tales sistemas de n ecuaciones lineales), estamos interesados en los métodos de factorización de la matriz de coeficientes como producto de matrices triangulares, que conduciría a la resolución sencilla de dos sistemas lineales con matriz triangular. Veamos algunos comandos de Maxima para realizar factorizaciones de matrices.

4.2.1. Factorización LU

Dada una matriz cuadrada A la factorización LU consiste en calcular tres matrices, una matriz permutación P, y otras dos matrices L y U, con L triangular inferior con unos en la diagonal, U triangular superior y tales que A = PLU. Maxima calcula dicha factorización directamente con el uso de comandos de la siguiente forma:

```
(%i55) load(linearalgebra)$

(%i56) A:matrix([1,0,2],[2,7,3],[2,1,0]);
```

(%o56) $$\begin{pmatrix} 1 & 0 & 2 \\ 2 & 7 & 3 \\ 2 & 1 & 0 \end{pmatrix}$$

```
(%i57) get_lu_factors(lu_factor(A));
```

(%o57) $$\left[\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 2 & \frac{1}{7} & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 2 \\ 0 & 7 & -1 \\ 0 & 0 & -\frac{27}{7} \end{pmatrix}\right]$$

Las tres matrices que aparecen son respectivamente P, L y U

```
(%i58) P:%o57[1];
```

(%o58) $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

```
(%i59) L:%o57[2];
```

(%o59) $\begin{pmatrix}1 & 0 & 0\\ 2 & 1 & 0\\ 2 & \frac{1}{7} & 1\end{pmatrix}$

```
(%i60) U:%o57[3];
```

(%o60) $\begin{pmatrix}1 & 0 & 2\\ 0 & 7 & -1\\ 0 & 0 & -\frac{27}{7}\end{pmatrix}$

Comprobamos que la factorización es correcta

```
(%i61) P.L.U;
```

(%o61) $\begin{pmatrix}1 & 0 & 2\\ 2 & 7 & 3\\ 2 & 1 & 0\end{pmatrix}$

Ha hecho la factorización A=PLU, pero sólo cambiando filas en caso de división por cero. Si queremos que haga pivoteo parcial tendremos que pasarle el argumento opcional floatfield. Notar que en este caso calcula la factorización de manera numérica

```
(%i62) get_lu_factors(lu_factor(A,floatfield));
```

(%o62) $\left[\begin{pmatrix}0 & 0 & 1\\ 1 & 0 & 0\\ 0 & 1 & 0\end{pmatrix}, \begin{pmatrix}1 & 0 & 0\\ 1{,}0 & 1 & 0\\ 0{,}5 & 0{,}58333333333333 & 1\end{pmatrix}, \begin{pmatrix}2{,}0 & 7{,}0 & 3{,}0\\ 0 & -6{,}0 & -3{,}0\\ 0 & 0 & 2{,}25\end{pmatrix}\right]$

```
(%i63) P:%o62[1];
```

(%o63) $\begin{pmatrix}0 & 0 & 1\\ 1 & 0 & 0\\ 0 & 1 & 0\end{pmatrix}$

```
(%i64) L:%o62[2];
```

(%o64) $\begin{pmatrix}1 & 0 & 0\\ 1{,}0 & 1 & 0\\ 0{,}5 & 0{,}58333333333333 & 1\end{pmatrix}$

```
(%i65) U:%o62[3];
```

(%o65) $\begin{pmatrix}2{,}0 & 7{,}0 & 3{,}0\\ 0 & -6{,}0 & -3{,}0\\ 0 & 0 & 2{,}25\end{pmatrix}$

```
(%i66) P.L.U;
```

(%o66) $\begin{pmatrix} 1{,}0 & 0{,}0 & 2{,}0 \\ 2{,}0 & 7{,}0 & 3{,}0 \\ 2{,}0 & 1{,}0 & 0{,}0 \end{pmatrix}$

Para resolver un sistema Ax=b utilizando la descomposición LU con pivoteo parcial, podemos seguir los pasos siguientes

```
(%i67) b:matrix([-1],[-1],[2]);
```

(%o67) $\begin{pmatrix} -1 \\ -1 \\ 2 \end{pmatrix}$

```
(%i68) M:lu_factor(A,floatfield);
```

(%o68) $[\begin{pmatrix} 0{,}5 & 0{,}58333333333333 & 2{,}25 \\ 2{,}0 & 7{,}0 & 3{,}0 \\ 1{,}0 & -6{,}0 & -3{,}0 \end{pmatrix}, [2,3,1], floatfield, 6{,}0, 80{,}8888888888889]$

```
(%i69) get_lu_factors(M);
```

(%o69) $[\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 1{,}0 & 1 & 0 \\ 0{,}5 & 0{,}58333333333333 & 1 \end{pmatrix}, \begin{pmatrix} 2{,}0 & 7{,}0 & 3{,}0 \\ 0 & -6{,}0 & -3{,}0 \\ 0 & 0 & 2{,}25 \end{pmatrix}]$

```
(%i70) lu_backsub(M,b);
```

(%o70) $\begin{pmatrix} 1{,}0 \\ -0{,}0 \\ -1{,}0 \end{pmatrix}$

4.2.2. Factorización de Cholesky

La factorización de Cholesky es una factorización parecida a la anterior, Maxima la realiza para matrices reales simétricas (o complejas hermíticas) y en este caso la matriz A se descompone en la forma $A = LL^t$ con L triangular inferior. Dicha factorización está asegurada para matrices reales simétricas y definidas positivas, siendo en este caso los elementos diagonales de L positivos. Maxima calcula dicha factorización directamente con el uso de comandos de la siguiente forma:

```
(%i71) A:matrix([2,-1,-2],[-1,14,7],[-2,7,5]);
```

(%o71) $\begin{pmatrix} 2 & -1 & -2 \\ -1 & 14 & 7 \\ -2 & 7 & 5 \end{pmatrix}$

```
(%i72) B:cholesky(A,floatfield);
```

(%o72) $\begin{pmatrix} 1{,}414213562373095 & 0 & 0 \\ -0{,}70710678118655 & 3{,}674234614174767 & 0 \\ -1{,}414213562373095 & 1{,}632993161855452 & 0{,}57735026918963 \end{pmatrix}$

4.2.3. Normas vectoriales y matriciales. Número de condición de una matriz

Vamos a calcular ahora las normas usuales de vectores y matrices con funciones de Maxima

```
(%i73) v:[1,-1,2];
```

(%o73) $[1, -1, 2]$

```
(%i74) lmax(abs(v)) /* norma infinito del vector v*/;
```

(%o74) 2

```
(%i75) v:matrix([1,-1,2]);
```

(%o75) $\begin{pmatrix} 1 & -1 & 2 \end{pmatrix}$

```
(%i76) mat_norm(v,1) /* norma 1 como matriz, coincide
       con la norma infinito como vector fila */;
```

(%o76) 2

```
(%i77) mat_norm(v,inf) /* norma infinito como matriz, coincide
       con la norma 1 como vector fila */;
```

(%o77) 4

```
(%i78) mat_norm(v,frobenius) /* norma 2 como vector */;
```

(%o78) $\sqrt{6}$

```
(%i79) A:matrix([1,-1],[2,3]);
```

(%o79) $\begin{pmatrix} 1 & -1 \\ 2 & 3 \end{pmatrix}$

```
(%i80) mat_norm(A,1)  /* norma 1 de la matriz A*/;
```

(%o80) 4

```
(%i81) mat_norm(A,inf) /* norma infinito de la matriz A */;
```

(%o81) 5

```
(%i82) mat_norm(A,frobenius) /* norma frobenius de la matriz A */;
```

(%o82) $\sqrt{15}$

No confundir esta con la norma dos de A, dada por

```
(%i83) norma_2(A):=sqrt(lmax(abs(eigenvalues(transpose(A).A)[1])));
```

(%o83) $\mathrm{norma}_2(A) := \sqrt{\mathrm{lmax}\left(\left|(\mathrm{eigenvalues}(\mathrm{transpose}(A).A))_1\right|\right)}$

```
(%i84) norma_2(A);
```

(%o84) $\dfrac{\sqrt{5^{\frac{3}{2}}+15}}{\sqrt{2}}$

Para vectores podemos definir directamente sus normas usuales por medio de las funciones siguientes

```
(%i85) norma_1(a):=apply("+",abs(a));
       norma_2(a):=sqrt(apply("+",abs(a)^2));
       norma_inf(a):=lmax(abs(a));
```

(%o85) $\mathrm{norma_1}(a) := \mathrm{apply}(+, |a|)$ (%o86) $\mathrm{norma_2}(a) := \sqrt{\mathrm{apply}\left(+, |a|^2\right)}$
(%o87) $\mathrm{norma_inf}(a) := \mathrm{lmax}(|a|)$

```
(%i88) a:[-5,3,4,-7];norma_1(a);norma_2(a);norma_inf(a);
```

(%o88) $[-5, 3, 4, -7]$ (%o89) 19 (%o90) $3\sqrt{11}$ (%o91) 7

En tanto que, para matrices cuadradas A, el radio espectral $\rho(A)$ (que denotaremos también por re(A)) y las normas matriciales compatibles con las anteriores, podemos obtenerlas con las siguientes:

```
(%i92) /*La siguiente función nos da el radio espectral, al
       menos, para matrices de orden menor o igual a cuatro*/
       re(A):=lmax(abs(eigenvalues(A)[1]))$
```

```
(%i93) norma_1(A):=block(
          s:matrix_size(A),n:s[1],
          for j:1 thru n do
          c[j]:sum(abs(A[i,j]),i,1,n),
          c:makelist(c[j],j,1,n),
          print("norma_1(A) = ",lmax(c))
          )$

(%i94) norma_2(A):=print("norma_2 (A) =",
       float(sqrt(re(transpose(A).A))))$

(%i95) norma_inf(A):=block(
          s:matrix_size(A),n:s[1],
          for i:1 thru n do
          f[i]:sum(abs(A[i,j]),j,1,n),
          f:makelist(f[i],i,1,n),
          print("norma_inf(A) = ",lmax(f))
          )$

(%i96) A:matrix([2,2,4],[2,3,6],[1,1,1]);
       print("Radio espectral de A =",float(re(A)))$
       norma_1(A)$ norma_2(A)$ norma_inf(A)$
```

(%o96) $\begin{pmatrix} 2 & 2 & 4 \\ 2 & 3 & 6 \\ 1 & 1 & 1 \end{pmatrix}$

$Radio\ espectral\ de\ A = 6{,}418832675970043$
$norma_1(A) = 11$
$norma_2(A) = 8{,}676601657378294$
$norma_inf(A) = 11$

Observación. Téngase en cuenta que la función "**eigenvalues**" llama a "**solve**" y puede que no sepa resolver muchas ecuaciones algebraicas de orden superior a 4, por ello en ese caso habrá que determinar el polinomio característico y resolverlo con otros comandos, por ejemplo "**algsys**".

Una vez tenemos controlado como calcular normas de matrices, vamos a pasar a calcular el número de condición de una matriz. Este número indica si la matriz está bien condicionada o no con respecto a la resolución de sistemas lineales. Un número de condición alto es malo, ya que indica posibles problemas en la resolución numérica de un sistema lineal que tenga a esa matriz como matriz de coeficientes. Los condicionamientos respecto de las normas usuales, están dados por las funciones siguientes:

```
(%i101) mat_cond(A,1) /* número de condición de la matriz A
        en norma 1 */;
```

(%o101) 44

```
(%i102) mat_cond(A,inf) /* número de condición de la matriz A
        en norma infinito */;
```

(%o102) 55

```
(%i103) mat_cond(A,frobenius) /* número de condición de la
        matriz A en norma frobenius */;
```

(%o103) $\sqrt{2}\,3^{\frac{3}{2}}\,\sqrt{19}$

Un ejemplo clásico de matrices mal condicionadas son las matrices de Vandermonde. Estas matrices aparecen de manera natural en el problema de interpolación de Lagrange. El número de condición de estas matrices crece con la dimensión, y eso es perjudicial.

```
(%i104) B:vandermonde_matrix([1,2,3,4,5]);
```

(%o104) $\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & 3 & 9 & 27 & 81 \\ 1 & 4 & 16 & 64 & 256 \\ 1 & 5 & 25 & 125 & 625 \end{pmatrix}$

```
(%i105) mat_cond(B,1);
```

(%o105) 44055

Vamos a resolver dos sistemas lineales con la anterior como matriz de coeficientes y con pequeños cambios en la columna de términos independientes, para ver como afecta a las soluciones

```
(%i106)kill(all)$
       linsolve([x+y+z+u+v=10,x+2*y+4*z+8*u+16*v=2,
       x+3*y+9*z+27*u+81*v=3,x+4*y+16*z+64*u+256*v=4,
       x+5*y+25*z+125*u+625*v=50],[x,y,z,u,v]),numer;

       /*Veamos ahora las soluciones del sistema perturbado,
       cambiando ligeramente los términos independientes de
       las ecuaciones anteriores*/

       linsolve([x+y+z+u+v=10.02,x+2*y+4*z+8*u+16*v=2.01,
       x+3*y+9*z+27*u+81*v=2.9,x+4*y+16*z+64*u+256*v=3.99,
       x+5*y+25*z+125*u+625*v=50.03],[x,y,z,u,v]),numer;
```

(%o1) $[x=90, y=-150{,}5, z=92{,}25, u=-24, v=2{,}25]$
(%o2) $[x=89{,}08, y=-148{,}6641666666667, z=91{,}09791666666666, u=-23{,}72083333333334, v=2{,}227083333333333]$

Ejercicio. Calcular A y B para que la función $f(x) = Ax^{12} + Bx^{13}$, verifique las condiciones:

1. $f(0{,}1) = 6{,}06 \cdot 10^{-13}$ y $f(0{,}9) = 0{,}03577$
2. $f(0{,}1) = 0{,}0001$ y $f(0{,}9) = 0{,}0356$

Para ello hemos de resolver los dos sistemas lineales siguientes.

```
(%i3)  kill(all)$ f(x):=A*x^12+B*x^13;
       linsolve([f(0.1)=6.06*10^-13,f(0.9)=0.03577],[A,B]),numer;

       /*Veamos otra vez las soluciones del sistema perturbado,
       cambiando ligeramente los términos independientes de las
       ecuaciones anteriores*/

       linsolve([f(0.1)=0.0001,f(0.9)=0.0356],[A,B]),numer;
```

(%o1) $\mathrm{f}(x) := A\,x^{12} + B\,x^{13}$
(%o2) $[A=0{,}66591861757552, B=-0{,}59918617575518]$
(%o3) $[A=1{,}1249999998424385\,10^{8}, B=-1{,}2499999984243855\,10^{8}]$

Para ver el porque de esta gran diferencia en las soluciones, por tan leve cambio en los términos independientes de estas ecuaciones, basta con ver el número de condición en norma 1 de la matriz de coeficientes de este sistema.

```
(%i4) coefmatrix([f(0.1)=6.06*10^-13,f(0.9)=0.03577],[A,B]),numer;
```

(%o4) $\begin{pmatrix} 9{,}9999999999999998\,10^{-13} & 1{,}0\,10^{-13} \\ 0{,}282429536481 & 0{,}2541865828329 \end{pmatrix}$

Y su número de condición en norma1 es

```
(%i5) mat_cond(%,1);
```

(%o5) $6{,}7077014914475159\,10^{11}$

4.3. Métodos iterativos de Jacobi y Gauss-Seidel.

4.3.1. Método de Jacobi

Dentro de los métodos iterativos para la resolución de sistemas de ecuaciones lineales, el más clásico es el método de Jacobi, cuya matriz de iteración viene dada por $E_J = D^{-1}(L + U)$ siendo convergente si y sólo si $\rho(E_J) < 1$. En particular, si A es estrictamente diagonal dominante el método es convergente. En las siguientes órdenes vamos a ver cómo aplicar dicho método en el supuesto de que sea convergente. Consideramos la matriz de coeficientes

```
(%i6) A:matrix([1,6,-1,2],[4,1,0,1],[0,-1,10,4],[3,-1,-2,7]);
```

(%o6) $\begin{pmatrix} 1 & 6 & -1 & 2 \\ 4 & 1 & 0 & 1 \\ 0 & -1 & 10 & 4 \\ 3 & -1 & -2 & 7 \end{pmatrix}$

Y el vector de términos independientes

```
(%i7) b:matrix([6],[6],[-2],[19]);
```

(%o7) $\begin{pmatrix} 6 \\ 6 \\ -2 \\ 19 \end{pmatrix}$

Debido a que es mejor trabajar con una matriz estrictamente diagonal dominante, ya que esto es condición suficiente para la convergencia de este método, vamos a intercambiar la primera con la segunda ecuación (que se traduce en el intercambio de esas filas en las matrices A y b)

```
(%i8)  A:rowswap(A,1,2);
```

(%o8) $\begin{pmatrix} 4 & 1 & 0 & 1 \\ 1 & 6 & -1 & 2 \\ 0 & -1 & 10 & 4 \\ 3 & -1 & -2 & 7 \end{pmatrix}$

```
(%i9)  b:rowswap(b,1,2);
```

(%o9) $\begin{pmatrix} 6 \\ 6 \\ -2 \\ 19 \end{pmatrix}$

Ahora inicializamos las variables, el número máximo de iteraciones "nmax", la dimensión s del sistema y los valores iniciales de las incógnitas, que tomamos iguales a cero

```
(%i10) n_max:100;
```

(%o10) 100

```
(%i11) s:matrix_size(b);
```

(%o11) [4, 1]

```
(%i12) s:s[1];
```

(%o12) 4

```
(%i13) P:zerofor(b);
```

(%o13) $\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$

```
(%i14) n:1;
```

(%o14) 1

```
(%i15) z:matrix([0],[0],[0],[0]);
```

(%o15) $\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$

```
(%i16) while (n<=nmax) do
         (
         for i:1 thru s do(
           z[i]:1/A[i,i]*(b[i]-sum(A[i,j]*P[j],j,1,i-1)
              -sum(A[i,j]*P[j],j,i+1,s))
                          ),
         P:z,
         n:n+1
         )$
```

Finalmente, mostramos la última aproximación obtenida de la solución

```
(%i17) print("La solución aproximada tras ", nmax, " pasos es ",
       float(z))$
```

La solución aproximada tras 100 pasos es $\begin{pmatrix} 1{,}0 \\ -2{,}8456248203415032\,10^{-62} \\ -1{,}0 \\ 2{,}0 \end{pmatrix}$

Que es una buena aproximación de la solución exacta dada por $x = (1, 0, -1, 2)$.

4.3.2. Método de Gauss-Seidel

Veamos seguidamente otro de los métodos iterativos usuales para la resolución de sistemas de ecuaciones lineales, el método de Gauss-Seidel, cuya matriz de iteración es $E_{GS} = (D - L)^{-1}U$ y converge si y sólo si $\rho(E_{GS}) < 1$. En particular, converge para sistemas lineales $Ax = b$ con A estrictamente diagonal dominante. Ahora presentamos un block para resolver un sistema por el método de Gauss-Seidel (en el supuesto de que sea convergente) con un número máximo de iteraciones "nmax"

```
(%i18) /* Método de Gauss-Seidel con nmax iteraciones para
          resolver un sistema de n ecuaciones lineales*/


       GS(A,b,nmax):=block([numer],numer:true,nmax,
         s:matrix_size(A),n:s[1],
         x: matrix([0],[0],[0],[0],[0],[0]),
         for k:1 thru nmax do
           (for i:1 thru n do(x[i]:(b[i]-sum(A[i,j]*x[j],j,1,i-1)
                              -sum(A[i,j]*x[j],j,i+1,n))/A[i,i])),
       print("La solución aproximada tras ",nmax," pasos es ",x))$
```

que ahora aplicamos al sistema lineal de seis ecuaciones dado por

```
(%i19) A: matrix([10,-3,2,1,1,0],[1,8,-1,3,0,1],[0,1,12,-5,0,2],
              [2,-2,3,20,1,1],[-2,-1,1,3,15,2],[1,-2,1,-1,1,9]);
       b: matrix([16],[-1],[0],[29],[38],[31]);
       GS(A,b,30)$
```

(%o19) $\begin{pmatrix} 10 & -3 & 2 & 1 & 1 & 0 \\ 1 & 8 & -1 & 3 & 0 & 1 \\ 0 & 1 & 12 & -5 & 0 & 2 \\ 2 & -2 & 3 & 20 & 1 & 1 \\ -2 & -1 & 1 & 3 & 15 & 2 \\ 1 & -2 & 1 & -1 & 1 & 9 \end{pmatrix}$

(%o20) $\begin{pmatrix} 16 \\ -1 \\ 0 \\ 29 \\ 38 \\ 31 \end{pmatrix}$

La solución aproximada tras 30 pasos es $\begin{pmatrix} 1{,}0 \\ -1{,}0 \\ 0{,}0 \\ 1{,}0 \\ 2{,}0 \\ 3{,}0 \end{pmatrix}$

Que es una buena aproximación a la solución exacta dada por $x = (1, -1, 0, 1, 2, 3)$.

Ejercicio. Mejorar el programa anterior para detener la ejecución si la

norma subinfinito del vector diferencia de dos iteraciones consecutivas es menor que un épsilon dado.

Tanto este método como el anterior pueden hacerse matricialmente, pero no es aconsejable, sobre todo para un gran número de ecuaciones e incógnitas.

4.4. Ejercicios propuestos

Completar con Maxima los ejercicios siguientes:

1. Utilizar los comandos generales de wxMaxima para resolver el sistema lineal: $x + 2y - z = -3,\ -x - 4y + z = 5,\ 2x + 10y + 2z = -4$.

2. Formar una matriz diagonal A con los valores $1, 2, 3, 4, 5, 6$ en su diagonal. Calcular después A^3 y hallar su traza (la suma de los elementos de su diagonal).

3. Construir funciones de Maxima para resolver sistemas triangulares, aplicarlas a algún ejemplo.

4. Realizar una factorización LU con pivoteo parcial de la matriz de coeficientes del sistema del ejercicio 1. Comprobar que dicha factorización está bien hecha, y además resolver usando la descomposición LU el sistema de ecuaciones lineales.

5. Realizar una factorización de Cholesky de la matriz

$$A = \begin{pmatrix} 1 & 2 & -1 \\ 2 & 8 & -2 \\ -1 & -2 & 17 \end{pmatrix}$$

   Comprobar que dicha factorización está bien hecha. Utilizar la factorización para resolver el sistema de ecuaciones $Ax = b$, siendo b el vector columna tal que $b^t = (0, 0, 16)$. Tendréis que resolver dos sistemas triangulares por sustitución progresiva y regresiva respectivamente.

6. Calcular el número de condición en norma 2 de la matriz A dada por

$$\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

7. Definir la matriz de Vandermonde de tamaño $n = 3, 4, 5$ cuando los nodos son $1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5$ respectivamente. Calcular el número de condición en norma 1 de dichas matrices. Podéis ver lo que sucede.

8. Hallar la matriz del método de Jacobi E_J cuando la matriz de coeficientes de un sistema es

$$A = \begin{pmatrix} 1 & 2 & -1 \\ 2 & 8 & -2 \\ -1 & -2 & 17 \end{pmatrix}$$

9. Calcular la matriz del método de Gauss-Seidel E_{GS} para la matriz A del apartado anterior.

10. Aplicar el método de Gauss-Seidel para aproximar la solución del sistema

$$\begin{array}{rcl} x_1 + 8x_2 - x_3 + 3x_4 + x_6 & = & -1 \\ 10x_1 - 3x_2 + 2x_3 + x_4 + x_5 & = & 16 \\ x_2 + 12x_3 - 5x_4 + 2x_6 & = & 0 \\ 2x_1 - 2x_2 + 3x_3 + 20x_4 + x_5 + x_6 & = & 29 \\ -2x_1 - x_2 + x_3 + 3x_4 + 15x_5 + 2x_6 & = & 38 \\ x_1 - 2x_2 + x_3 - x_4 + x_5 + 9x_6 & = & 31 \end{array}$$

Realizar $n = 20$ iteraciones. Mejorarlo, si podéis, para que en vez de con el número máximo de iteraciones, el programa pare cuando la distancia entre dos iterados sucesivos sea menor que 10^{-5}.

11. Resolver el problema de contorno que sigue:

$$y''(x) = sin(x),\ x \in [0,1];\ y(0) = y(1) = 0$$

por diferencias finitas, para ello considerar las $n = 10$ ecuaciones lineales que salen de discretizar la segunda derivada en la forma

$$y''(x_j) = \frac{y_{j-1} - 2y_j + y_{j+1}}{h^2}$$

siendo $h = \frac{1}{n}$, $y_j = y(x_j)$ y $x_j = jh$. Dibujar la gráfica de la solución.

Capítulo 5

Interpolación, derivación e integración numérica. Mejor aproximación por mínimos cuadrados

5.1. Interpolación

5.1.1. Cálculo directo del polinomio de interpolación

Como es sabido, dados los valores de una función $f(x)$, $f_i = f(x_i)$ en $n + 1$ puntos distintos $\{x_i|(i = 0, 1, 2, ..., n)\}$ del intervalo $[a, b]$, existe un único polinomio $p(x)$ de grado menor o igual a n tal que $p(x_i) = f_i$ para cada $i = 0, 1, 2, \cdots, n$, a dicho polinomio se le llama el polinomio de interpolación de f en los $n + 1$ puntos dados. Puede escribirse como $P(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0$ y hallar los coeficientes $a_0, a_1, ..., a_n$ del polinomio de interpolación de manera que se cumplan las $n + 1$ ecuaciones lineales:

$$P(x_i) = f_i \ (i = 0, 1, \cdots, n)$$

siendo el sistema correspondiente compatible determinado, pues el determinante de la matriz de coeficientes es un determinante de Vandermonde, que no se anula por ser los puntos distintos, aunque sabemos que la matriz está mal condicionada, de ahí el interés en otros métodos para obtener el polinomio de interpolación. Pero veamos algún ejemplo con este método. Se desea obtener el polinomio de interpolación que pasa por los puntos: $(0, 1), (1, 5), (2, 31), (3, 121), (4, 341)$, como se trata de 5 puntos por ellos pasa un único polinomio de grado, a lo más, cuatro de la forma

$P(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4$, para hallarlo hemos de resolver el sistema lineal:

```
(%i1) p(x):=a0+a1*x+a2*x^2+a3*x^3+a4*x^4;
      linsolve([p(0)=1,p(1)=5,p(2)=31,
      p(3)=121,p(4)=341],[a0,a1,a2,a3,a4]);
```

(%o1) $\mathrm{p}(x) := a0 + a1\,x + a2\,x^2 + a3\,x^3 + a4\,x^4$
(%o2) $[a0 = 1, a1 = 1, a2 = 1, a3 = 1, a4 = 1]$

Luego se trata del polinomio $p(x) = x^4 + x^3 + x^2 + x + 1$.

5.1.2. Fórmula de Lagrange

La fórmula de Lagrange está dada por

$$p(x) = f_0l_0(x) + f_1l_1(x) + ... + f_nl_n(x)$$

donde $fi = f(x_i)$ y $l_i(x)$ son los polinomios de base de Lagrange. Para el ejemplo del párrafo anterior se tendrá

```
(%i3) p(x):=(x-1)*(x-2)*(x-3)*(x-4)/((-1)*(-2)*(-3)*(-4))+
      5*((x*(x-2)*(x-3)*(x-4))/((1-0)*(1-2)*(1-3)*(1-4)))+
      31*((x*(x-1)*(x-3)*(x-4))/((2-0)*(2-1)*(2-3)*(2-4)))+
      121*((x*(x-1)*(x-2)*(x-4))/((3-0)*(3-1)*(3-2)*(3-4)))
      +341*((x*(x-1)*(x-2)*(x-3))/((4-0)*(4-1)*(4-2)*(4-3)));
      expand(p(x));
```

(%o3) $\mathrm{p}(x) := \frac{(x-1)\,(x-2)\,(x-3)\,(x-4)}{(-1)\,(-2)\,(-3)\,(-4)} +$
$+ 5\,\frac{x\,(x-2)\,(x-3)\,(x-4)}{(1-0)\,(1-2)\,(1-3)\,(1-4)} + 31\,\frac{x\,(x-1)\,(x-3)\,(x-4)}{(2-0)\,(2-1)\,(2-3)\,(2-4)} +$
$+ 121\,\frac{x\,(x-1)\,(x-2)\,(x-4)}{(3-0)\,(3-1)\,(3-2)\,(3-4)} + 341\,\frac{x\,(x-1)\,(x-2)\,(x-3)}{(4-0)\,(4-1)\,(4-2)\,(4-3)}$
(%o4) $x^4 + x^3 + x^2 + x + 1$

Resultando, lógicamente, el polinomio hallado antes.

5.1.3. Fórmula de Newton en diferencias divididas

La fórmula de Newton, en diferencias divididas, del polinomio de interpolación está dada por

$$p(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + ...$$

$$+f[x_0, x_1, ..., x_n](x - x_0)(x - x_1)...(x - x_{n-1})$$

siendo las $f[x_0, x_1, ..., x_i]$ las diferencias divididas de f de orden i en los puntos $x_0, x_1, \cdots, x_i$, que se calculan recurrentemente en la forma:

$$f[x_0, x_1, ..., x_i] = (f[x_1, ..., x_i] - f[x_0, x_1, ..., x_{i-1}])/(x_i - x_0)$$

siendo $f[x_k] = f(x_k) = fk$. En tanto que, las diferencias divididas $f[x_0, x_1, ..., x_k]$ las escribimos en lo que sigue como $f01 \cdots k$.

```
(%i5)  kill(all)$ x0:0 $x1:1 $x2:2 $x3:3 $x4:4$
       f0:1$ f1:5$ f2:31$ f3:121$ f4:341$
       f01:(f1-f0)/(x1-x0)$
       f12:(f2-f1)/(x2-x1)$
       f23:(f3-f2)/(x3-x2)$
       f34:(f4-f3)/(x4-x3)$
       f012:(f12-f01)/(x2-x0)$
       f123:(f23-f12)/(x3-x1)$
       f234:(f34-f23)/(x4-x2)$
       f0123:(f123-f012)/(x3-x0)$
       f1234:(f234-f123)/(x4-x1)$
       f01234:(f1234-f0123)/(x4-x0)$
       p(x):=f0+f01*(x-x0)+f012*(x-x0)*(x-x1)+
       f0123*(x-x0)*(x-x1)*(x-x2)+
       f01234*(x-x0)*(x-x1)*(x-x2)*(x-x3);
       print("p(x) = ",expand(p(x))$
```

(%o22) $p(x) = x^4 + x^3 + x^2 + x + 1$

Que vuelve a dar el polinomio de interpolación que hemos hallado antes.

5.1.4. Taylor

Si tenemos una función $f(x)$, derivable hasta el orden n en un entorno del punto $x = a$, podemos aproximarla, en las proximidades de a, por un polinomio. El criterio con el que elegiremos el polinomio será hacer coincidir la función y sus derivadas sucesivas con los correspondientes valores del polinomio en el punto $x = a$. Este problema tiene solución única dada por el polinomio de Taylor de orden n de una función f en el punto a. En Maxima puede obtenerse con el siguiente comando hasta el orden que se desee.

```
(%i1)  taylor(f(x),x,a,5);
```

(%o1)/T/

$$f(a) + (\frac{d}{dx}f(x)|_{x=a})(x-a) + \frac{(\frac{d^2}{dx^2}f(x)|_{x=a})(x-a)^2}{2} + \frac{\left(\frac{d^3}{dx^3}\,\mathrm{f}(x)\Big|_{x=a}\right)(x-a)^3}{6} +$$
$$+ \frac{\left(\frac{d^4}{dx^4}\,\mathrm{f}(x)\Big|_{x=a}\right)(x-a)^4}{24} + \frac{\left(\frac{d^5}{dx^5}\,\mathrm{f}(x)\Big|_{x=a}\right)(x-a)^5}{120} + \ldots$$

Por ejemplo, si deseamos el polinomio de Taylor de grado 6 de la función $cos(x)$ alrededor de $x = 0$ pondremos:

```
(%i2)  taylor(cos(x),x,0,6);
```

(%o2)/T/
$$1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \ldots$$

La orden "**trunc**" guarda el polinomio de Taylor (cosas internas de máxima) como una función normal. Esta manera de guardarlo es más cómoda para posteriores evaluaciones.

```
(%i3)  trunc(taylor(cos(x),x,0,6));
```

(%o3) $1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \ldots$

En tanto que, el polinomio de Taylor de grado 5 de la función $cos(x)$ alrededor de $x = \pi$ será

```
(%i4)  taylor(cos(x)/x,x,%pi,5);
```

(%o4)/T/ $-\frac{1}{\pi} + \frac{x-\pi}{\pi^2} + \frac{\left(\pi^2-2\right)(x-\pi)^2}{2\,\pi^3} - \frac{\left(\pi^2-2\right)(x-\pi)^3}{2\,\pi^4} - \frac{\left(\pi^4-12\,\pi^2+24\right)(x-\pi)^4}{24\,\pi^5} + \frac{\left(\pi^4-12\,\pi^2+24\right)(x-\pi)^5}{24\,\pi^6} + \cdots$

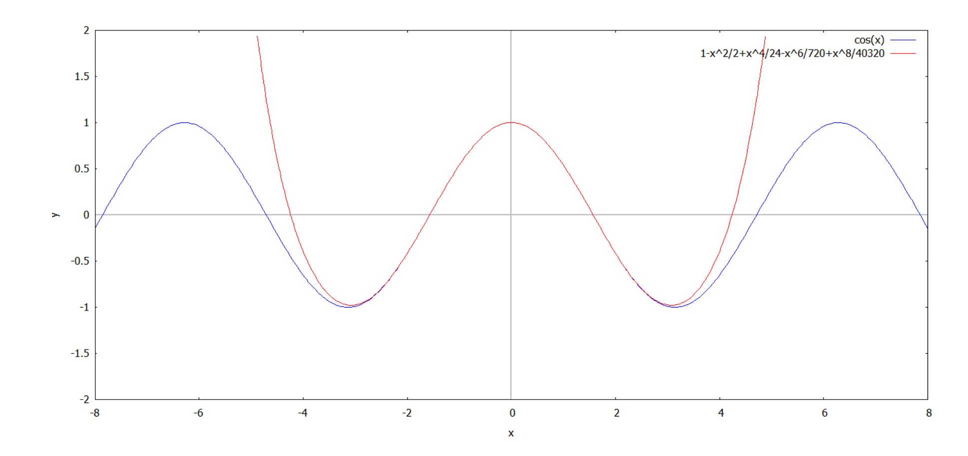
Dibujemos la función $f(x) = cos(x)$ y su polinomio de Taylor de orden 8 en torno a $x = 0$

```
(%i5)  f(x):=cos(x);
```

(%o5) $\mathrm{f}(x) := \cos(x)$

```
(%i6)  plot2d([f(x),taylor(f(x),x,0,8)],[x,-8,8],[y,-2,2]);
```

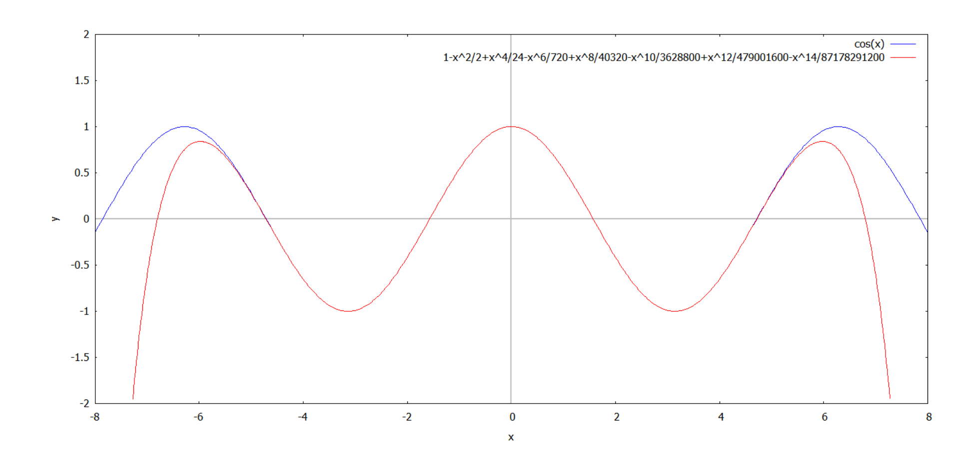
(%o6)



En teoría, un polinomio de Taylor de orden más alto debería aproximar mejor a la función. Vamos ahora a dibujar las gráficas de la función $f(x) = cos(x)$ y de su polinomio de Taylor de orden 14 en el cero para comprobar que la aproximación es más exacta.

```
(%i7)  plot2d([f(x),taylor(f(x),x,0,14)],[x,-8,8],[y,-2,2]);
```

(%o7)



Ejercicio. Programe con Maxima lo siguiente: Sea la función $f(x) = 1 + cos(x)$ ¿Cuál es el grado mínimo n para que el polinomio de Taylor $T(x)$ en torno a $x = 0$ cumpla que $|f(0{,}3) - T(0{,}3)| \leq 0{,}0001$?

```
(%i8)  k:1;
```

(%o8) 1

```
(%i9) f(x):=1+cos(x);
```

(%o9) $\mathrm{f}(x) := 1 + \cos(x)$

```
(%i10) define(T(x),trunc(taylor(f(x),x,0,k)));
```

(%o10) $\mathrm{T}(x) := 2$

```
(%i11) while abs(f(0.3)-T(0.3))>0.0001 do
       (k:k+1,define(T(x),trunc(taylor(f(x),x,0,k))));
```

(%o11) *done*

```
(%i12) k;
```

(%o12) 4

5.1.5. Splines Naturales

De acuerdo con lo visto en teoría, primero definimos los nodos de interpolación

```
(%i13) t1:1;
```

(%o13) 1

```
(%i14) t2:2;
```

(%o14) 2

```
(%i15) t3:5;
```

(%o15) 5

Segundo, definimos los valores de la función en los nodos

```
(%i16) f1:-1;
```

(%o16) -1

```
(%i17) f2:1;
```

(%o17) 1

```
(%i18) f3:3;
```

(%o18) 3

En tercer lugar, calculamos los espaciados entre nodos

```
(%i19) h1:t2-t1;
```

(%o19) 1

```
(%i20) h2:t3-t2;
```

(%o20) 3

En cuarto lugar, especificamos que son splines naturales

```
(%i21) z1:0; z3:0;
```

(%o21) 0 (%o22) 0

Luego, resolvemos el sistema lineal para calcular las z's (derivadas segundas en los nodos intermedios)

```
(%i23) derivadas_segundas: linsolve([h1*z1+2*(h1+h2)*z2+h2*z3=
       6/h2*(f3-f2)-6/h1*(f2-f1)],[z2]);
```

(%o23) $[z2 = -1]$

```
(%i24) z2:rhs(derivadas_segundas[1]);
```

(%o24) -1

Para acabar dando las expresiones de cada uno de los polinomios cúbicos de interpolación, comenzando por el primero

```
(%i25) S1(x):=z2/(6*h1)*(x-t1)^3+z1/(6*h1)*(t2-x)^3+
       (f2/h1-z2*h1/6)*(x-t1)+(f1/h1-z1*h1/6)*(t2-x);
```

(%o25) $\mathrm{S1}\left(x\right) := \frac{z2}{6\,h1}\,\left(x-t1\right)^3+\frac{z1}{6\,h1}\,\left(t2-x\right)^3+\left(\frac{f2}{h1}-\frac{z2\,h1}{6}\right)\,\left(x-t1\right)+\left(\frac{f1}{h1}-\frac{z1\,h1}{6}\right)\,\left(t2-x\right)$

Damos la expresión del segundo polinomio (en este ejemplo no hay más)

```
(%i26) S2(x):=z3/(6*h2)*(x-t2)^3+z2/(6*h2)*(t3-x)^3+
       (f3/h2-z3*h2/6)*(x-t2)+(f2/h2-z2*h2/6)*(t3-x);
```

(%o26) $\mathrm{S2}\left(x\right) := \frac{z3}{6\,h2}\,\left(x-t2\right)^3+\frac{z2}{6\,h2}\,\left(t3-x\right)^3+\left(\frac{f3}{h2}-\frac{z3\,h2}{6}\right)\,\left(x-t2\right)+\left(\frac{f2}{h2}-\frac{z2\,h2}{6}\right)\,\left(t3-x\right)$

Ahora utilizamos los polinomios hallados para hacer estimaciones, por ejemplo para aproximar $f(1{,}5)$ utilizaremos la primera cúbica $S1(x)$.

```
(%i27) S1(1.5);
```

(%o27) 0,0625

Y si queremos estimar $f'(2)$, podemos utilizar la primera o la segunda, pues 2 es un nodo de interpolación.

```
(%i28) define(dS1(x),diff(S1(x),x));
```

(%o28) $\mathrm{dS1}\left(x\right) := \frac{13}{6} - \frac{{\left(x-1\right)}^{2}}{2}$

```
(%i29) dS1(2);
```

(%o29) $\frac{5}{3}$

Si derivamos $S2(x)$ debe salir lo mismo, ya que $t = 2$ es un nodo de interpolación

```
(%i30) define(dS2(x),diff(S2(x),x));
```

(%o30) $\mathrm{dS2}\left(x\right) := \frac{{\left(5-x\right)}^{2}}{6} + \frac{1}{6}$

```
(%i31) dS2(2);
```

(%o31) $\frac{5}{3}$

Para estimar $f''(2)$ podemos utilizar $S1(x)$ o $S2(x)$, por la razón aducida.

```
(%i32) define(d2S1(x),diff(S1(x),x,2));
```

(%o32) $\mathrm{d2S1}\left(x\right) := 1 - x$

```
(%i33) d2S1(2);
```

(%o33) -1

```
(%i34) define(d2S2(x),diff(S2(x),x,2));
```

(%o34) $\mathrm{d2S2}\left(x\right) := -\frac{5-x}{3}$

```
(%i35) d2S2(2);
```

(%o35) -1

5.1.6. El paquete "interpol"

Maxima dispone del paquete "interpol" que permite interpolar un conjunto de datos por una función lineal a trozos es decir por la poligonal que pasa por ellos, por un polinomio o por splines cúbicos, para ello se comienza cargando dicho paquete mediante la orden "**load(interpol)**", luego los datos a interpolar, supongamos que son los dados antes, mediante "datos:[[0, 1],[1, 5],[2, 31],[3, 121],[4, 341]]" y finalmente la salida requerida se pide en la forma:

- **linearinterpol(datos)** si se requiere la función poligonal que pasa por los puntos dados.
- **lagrange(datos)** si se requiere el polinomio de interpolación de Lagrange que pasa por esos puntos.
- **spline(datos)** si se requiere el spline cúbico natural con esos datos.

Ejercicio. Aplicar estos comandos para obtener el polinomio de interpolación de Lagrange para los datos anteriores y para obtener el spline cúbico del párrafo (5.1.5).

5.1.7. Problemas mixtos de interpolación

Hay problemas que no se ajustan a ninguno de los métodos típicos que se estudian en un curso de cálculo numérico básico, tales como Lagrange, Hermite, Splines, etcétera. Para resolver algunos de estos problemas no hay mas que resolver sistemas lineales. Pongamos por ejemplo que queremos buscar un polinomio de grado 2 que satisfaga $p(1) = 2, p'(1) = -1, p'(2) = 3$. Los coeficientes del polinomio serán las soluciones del sistema lineal

```
(%i36) coef:linsolve([a+b+c=2,2*a+b=-1,4*a+b=3],[a,b,c]);
```

(%o36) $[a = 2, b = -5, c = 5]$

Así el polinomio de grado 2 que satisface las condiciones anteriores es $p(x) = 2x^2 - 5x + 5$.

5.2. Fórmulas de derivación numérica de tipo interpolatorio

Estas fórmulas consisten en aproximar las derivadas de una función en un punto por las derivadas correspondientes de su polinomio de interpolación

en unos puntos dados, también pueden obtenerse imponiendo condiciones de exactitud, que se traducen en un sistema lineal en los coeficientes de la fórmula.

Supongamos que conocemos los valores de una función derivable en los puntos x_0, $x_1 = x_0+h$ y $x_2 = x_0+2h$ y queremos obtener un valor aproximado de $f'(x_0)$ mediante una fórmula del tipo

$$f'(x_0) \simeq (c_0 f(x_0) + c_1 f(x_1) + c_2 f(x_2))/h$$

que sea exacta de grado 2. Para ello, basta imponer que dicha fórmula sea exacta para $f(x) = 1, x, x^2$, lo que se traduce en resolver el siguiente sistema lineal en los coeficientes c_0, c_1, c_2

```
(%i37) coef:linsolve([c0+c1+c2=0,c1+2*c2=1,c1/2+2*c2=0],[c0,c1,c2]);
```

(%o37) $[c0 = -\frac{3}{2}, c1 = 2, c2 = -\frac{1}{2}]$

Luego se trata de la fórmula de derivación numérica

$$f'(x_0) \simeq \frac{-3f(x_0) + 4f(x_0+h) - f(x_0+2h)}{2h}$$

Para probar la inestabilidad del problema de derivación numérica, utilizaremos la fórmula de derivación numérica de tipo interpolatorio de primer orden progresiva: $f'(x) \simeq (f(x+h) - f(x))/h$ para aproximar la derivada de $f(x) = arctg(x)$ en el punto $x = \sqrt{2}$, como es sabido esta derivada vale $1/3$. En este caso, ejecutando el siguiente programa, en el que i es el número de la iteración, d muestra el numerador o incremento de la función $f(x+h)-f(x)$, h el incremento de la variable independiente, ad es la aproximación de la derivada y “error” muestra el error de la aproximación ad obtenida. Viendo la salida que da for que omitimos, se observa que la mejor aproximación se obtiene para $i = 25$, siendo $h = 2{,}9802322387695313 \cdot 10^{-8}$, luego empiezan a empeorar los resultados debido a la cancelación de cifras significativas y a los errores de redondeo.

```
(%i38) numer:true$ f(x):=atan(x); a:sqrt(2);h:1;N:50;
       for i:0 thru 50 do(F:f(a+h),d:F-f(a), ad:d/h,h:h/2,
       print("i = ",i," d = ",d, " h = ",2*h,
       " ad = ",ad, " error = ",1/3-ad))$
```

(%o39) $\mathrm{f}(x) := \mathrm{atan}(x)$ (%o40) 1,414213562373095
(%o41) 1 (%o42) 50

Ahora utilizaremos la fórmula de derivación numérica de tipo interpolatorio centrada de segundo orden: $f'(x) \simeq (f(x+h) - f(x-h))/2h$ para

aproximar la derivada de $f(x) = arctg(x)$ en el punto $x = \sqrt{2}$. En este caso ejecutando el siguiente programa, se observa que la mejor aproximación se obtiene para $i = 17$, siendo $h = 7{,}62939453125 \cdot 10^{-6}$, luego empiezan a empeorar los resultados debido a la cancelación de cifras significativas y a los errores de redondeo.

```
(%i44) numer:true$ f(x):=atan(x); a:sqrt(2);h:1;N:26;
       for i:0 thru 50 do(F2:f(a+h),F1:f(a-h),d:F2-F1,
       ad:d/(2*h),h:h/2, print("i = ",i,"   d = ",d,
       "   h = ",2*h, "   ad = ",ad,
       "   error = ",1/3-ad))$
```

(%o45) $\mathrm{f}(x) := \mathrm{atan}(x)$ (%o46) 1,414213562373095
(%o47) 1 (%o48) 26

En este caso la mejora se debe a que el método es de segundo orden, con error de truncatura de la forma $O(h^2)$, no como el anterior que es de la forma $O(h)$.

5.3. Fórmulas de integración numérica de tipo interpolatorio

Son las que se obtienen sustituyendo la función integrando por su polinomio de interpolación en puntos adecuados del intervalo de integración. Si los puntos están igualmente espaciados, las fórmulas se dicen de Newton-Côtes, cerradas si entran los extremos del intervalo y abiertas en otro caso. Cuando el intervalo de integración se divide en un número finito de subintervalos de la misma longitud y sobre cada uno de ellos se utiliza una fórmula de integración numérica de tipo interpolatorio simple, se obtiene una fórmula compuesta, veamos seguidamente dos de las más usuales.

5.3.1. Fórmula del trapecio compuesta

Se trata ahora de dividir el intervalo de integración $[a, b]$ en n partes iguales, de amplitud $h = (b - a)/n$, y aplicar a cada una de ellas la conocida regla del trapecio, que aproxima la integral de f en cada subintervalo $[x_i, x_{i+1}]$ en la forma

$$\int_{x_i}^{x_{i+1}} f(x) \simeq (h/2)[f(x_i) + f(x_{i+1}]$$

Por ejemplo, si queremos aproximar la integral de $f(x) = cos(x^2)$ en el intervalo $[0, 1]$, comenzamos definiendo el número n de veces que se aplica la

fórmula del trapecio simple

```
(%i50) n:100;
```

(%o50) 100

Ahora, definimos la función a integrar

```
(%i51) f(x):=cos(x^2);
```

(%o51) $\mathrm{f}(x) := \cos\left(x^2\right)$

Definimos E como la suma de los valores de la función en los puntos extremos

```
(%i52) E:f(0)+f(1);
```

(%o52) 1,54030230586814

Espaciado y nodos interiores

```
(%i53) a:0;
```

(%o53) 0

```
(%i54) b:1;
```

(%o54) 1

```
(%i55) h:(b-a)/n;
```

(%o55) 0,01

```
(%i56) for i:1 thru n-1 do(
       x[i]:a+i*h
       );
```

(%o56) *done*

```
(%i57) x[1];
```

(%o57) 0,01

```
(%i58) x[2];
```

(%o58) 0,02

```
(%i59) x[n-1];
```

(%o59) 0,99

Calculamos la contribución de los nodos interiores

```
(%i60) I:0;
```

(%o60) 0

```
(%i61) for i:1 thru n-1 do(I:I+f(x[i]));
```

(%o61) *done*

```
(%i62) I,numer;
```

(%o62) 89,68087018510771

Finalmente, obtenemos el valor aproximado de la integral en la forma

```
(%i63) V:h*(E/2+I),numer;
```

(%o63) 0,90451021338042

Ahora, reuniendo todo lo anterior, construiremos una función de Maxima, mediante un block, para el cálculo aproximado de integrales mediante la regla del trapecio compuesta como sigue:

```
(%i64) TC(f,a,b,n):=block([numer],numer:true,
       E:f(a)+f(b),h:(b-a)/n,for i:1 thru n-1 do(x[i]:a+i*h),
       I:0,for i:1 thru n-1 do (I:I+f(x[i])),
       print('integrate(f(x),x,a,b),"~",h*(E/2+I)))$
```

Seguidamente la aplicamos para aproximar la integral de $f(x) = cos(x^2)$ en el intervalo $[0, 1]$

```
(%i65) f(x):=cos(x^2)$ TC(f,0,1,100)$
```

$$\int_0^1 \cos\left(x^2\right) dx \simeq 0{,}90451021338042$$

Observación. Puesto que conocemos la expresión del error de la fórmula del trapecio compuesta, caso de ser la función a integrar de clase $\mathcal{C}^{(2)}$ y de que se pueda acotar el valor absoluto de $f''(x)$ en el intervalo dado, puede mejorarse el programa, obteniendo previamente el número n de subdivisiones a realizar para conseguir que el módulo del error absoluto cometido sea menor que un épsilon dado.

5.3.2. Regla de Simpson compuesta

En este caso utilizaremos la regla de Simpson compuesta para aproximar una integral, lo vemos paso a paso en el siguiente ejemplo, en el que además calculamos primero en cuántos subintervalos iguales hay que dividir el intervalo de integración para conseguir una aproximación con error absoluto menor que una épsilon dado. La fórmula del trapecio simple aproxima una integral en la forma

$$\int_c^d f(x) \simeq \frac{d-c}{6}[f(c)+4f(\frac{c+d}{2})+f(d)]$$

Aproximar utilizando la fórmula de Simpson compuesta el área definida entre la curva $y = 1/(5-x)$, $2 \leq x \leq 3$ y el eje x. ¿Cuántas divisiones del intervalo $[2, 3]$ son necesarias para asegurar un error menor que 10^{-6}?.

```
(%i67) kill(all)$ numer:true$ f(x):=1/(5-x); a:2; b:3;
```

(%o2) $\mathrm{f}(x) := \frac{1}{5-x}$

(%o3) 2

(%o4) 3

Introducimos la cota de error

```
(%i5) Er:10^-6;
```

(%o5) $9{,}9999999999999995\,10^{-7}$

Para obtener una cota del error absoluto de la regla de Simpson compuesta, hacemos la derivada cuarta de $f(x)$, que resulta ser

```
(%i6) f4:diff(f(x),x,4);
```

(%o6) $\frac{24}{(5-x)^5}$

Con lo cual el máximo valor absoluto de la misma en $[2, 3]$ es

```
(%i7) f4m:24/(5-3)^5;
```

(%o7) 0,75

Por tanto, hemos de dividir el intervalo en un número n de subintervalos mayor que el número siguiente (ver por ejemplo en [10]).

```
(%i8) ((b-a)^5*f4m/(2880*Er))^0.25;
```

(%o8) 4,017142094723259

Luego, para estar seguros tomamos $n = 5$.

```
(%i9)  n:5;
```

(%o9) 5

Subdividimos el intervalo de partida en $2 \times 5 = 10$ partes, definimos el paso h entre nodos como $(b-a)/2n$ y con objeto de considerar la aportación de los puntos extremos (Ext), los que separan intervalos (Ip) y los puntos medios (Ii), inicializamos como cero estas cantidades.

```
(%i10) Ext:0;Ip:0;Ii:0;h:(b-a)/(2*n);
```

(%o10) 0
(%o11) 0
(%o12) 0
(%o13) 0,1

Ahora definimos los nodos

```
(%i13) h:(b-a)/(2*n);
```

(%o13) 0,1

```
(%i14) for i:0 thru 2*n do(x[i]:2+i*h);
```

(%o14) *done*

Calculemos la contribución de los nodos de índice par, que tal y como los hemos numerado son los puntos que separan subintervalos en los que se aplica la regla de Simpson simple

```
(%i15) for i:2 step 2 thru 2*n do (Ip:Ip+f(x[i]));
```

(%o15) *done*

Calculemos la contribución de los nodos de índice impar, que son los puntos medios de los subintervalos en los que se aplica la fórmula de integración simple, que resulta ser

```
(%i16) for i:1 step 2 thru 2*n-1 do (Ii:Ii+f(x[i]));
```

(%o16) *done*

Y finalmente la de los puntos extremos

```
(%i17) Ext:f(x[0])+f(x[2*n]);
```

(%o17) 0,83333333333333

Aplicando la regla de Simpson compuesta, obtenemos la aproximación pedida

```
(%i18) area_aprox: h/3*(Ext+4*Ii+2*Ip),numer;
```

(%o18) 0,40546527417092

Calculemos el área exacta mediante la regla de Barrow

```
(%i19) area_exacta:integrate(f(x),x,2,3),numer;
```

(%o19) 0,40546510810816

Con lo que la diferencia entre la solución aproximada y exacta será

```
(%i20) error_aprox: abs(area_aprox-area_exacta);
```

(%o20) $1{,}6606275687891525\,10^{-7}$

Generalizando lo anterior, definiremos otra función de Maxima, mediante un block, para el cálculo aproximado de integrales mediante la regla de Simpson compuesta; para ello, consideramos la función f en el intervalo $[a, b]$, definimos una partición del intervalo de amplitud $h = (b-a)/2n$, dividiendo el intervalo en $2n$ partes iguales y aplicamos la fórmula de Simpson simple n veces, una a cada subintervalo $[x_i, x_{i+2}]$ para $i = 0, 2, ..., 2(n-1)$, siendo $x_0 = a, x_i = x_0+ih$ $(i = 1, 2, ..., 2n)$ y $x_{2n} = b$; puesto que la amplitud de cada uno de los n subintervalos es $2h$ resulta que $2h/6 = h/3$, por tanto la integral de f en cada subintervalo se aproxima por $(h/3)[f(x_i) + 4f(x_{i+1}) + f(x_{i+2}]$; entonces, denotando la contribución de los puntos extremos por Ext, la de los puntos intermedios de índice impar por Ii (estos son los puntos medios de los intervalos donde se aplica la regla de Simpson simple) y por Ip la correspondiente a los puntos intermedios de índice par (estos son los puntos que separan los subintervalos donde se aplica la fórmula de Simpson simple), resultará la fórmula de Simpson compuesta dada por la siguiente función.

```
(%i21) SC(f,a,b,n):=block([numer],numer:true,
       Ext:f(a)+f(b),h:(b-a)/(2*n),for i:0 thru 2*n
       do(x[i]:a+i*h),
       Ii:0,for i:1 step 2 thru 2*n-1 do (Ii:Ii+f(x[i])),
       Ip:0, for i:2 step 2 thru 2*n-2 do (Ip:Ip+f(x[i])),
       print('integrate(f(x),x,a,b),"(con SC)~",
       (h/3)*(Ext+4*Ii+2*Ip)))$
```

Aplicando dicha regla al problema del párrafo anterior

```
(%i22) f(x):=cos(x^2)$ SC(f,0,1,100)$
```

$$\int_0^1 \cos\left(x^2\right) dx(conSC) \simeq 0{,}90452423790113$$

Observación. Al igual que en el caso anterior, dado que conocemos la expresión del error de la fórmula de Simpson compuesta, caso de ser la función a integrar de clase $\mathcal{C}^{(4)}$ y de que se pueda acotar el valor absoluto de $f^{(iv)}(x)$ en el intervalo dado, puede mejorarse el programa, obteniendo previamente el número n de subdivisiones a realizar para conseguir que el módulo del error absoluto cometido sea menor que un épsilon prefijado, evitando cálculos superfluos.

Ejercicio. Aplicar las fórmulas del trapecio y de Simpson compuestas a la función $f(x) = 1/(5-x)$ en el intervalo $[2,3]$, con $n = 100$. Comparar los resultados con el valor exacto de la misma dado por $log(3/2) = 0{,}40546510810816$.

5.4. Mejor aproximación por mínimos cuadrados continua o discreta

Sea $V = \mathcal{C}([a,b])$ el espacio vectorial real de las funciones continuas en el intervalo cerrado y acotado $[a,b]$, si para cada par de funciones f y g de V, definimos el producto escalar en la forma $(f|g) = \int_a^b f(x)g(x)dx$, tendremos un espacio prehilbertiano real. Entonces, dado cualquier subespacio S de V, de dimensión finita m, para toda $f \in V$ existe un único $f_s \in S$ tal que $f - f_s \in S^{\perp}$, es decir existe un único $f_s \in S$ tal que $(f - f_s|\nu) = 0$ para todo $\nu \in S$ y, teniendo en cuenta la definición del producto escalar, podemos decir que para toda $f \in V$ existe un único $f_s \in S$ tal que $\int_a^b (f(x) - f_s(x))\nu(x)dx = 0$ para toda $\nu(x) \in S$, siendo esta f_s **la mejor aproximación por mínimos cuadrados continua** de f por elementos de S.La mejor aproximación f_s puede hallarse reolviendo el sistema lineal de mínimos cuadrados:

- Si $\{\nu_1, \nu_2, \cdots, \nu_m\}$ es una base de S, $f_s = \sum_{i=1}^m \lambda_i \nu_i$ y se calculan los $\lambda_1, \lambda_2, \cdots, \lambda_m$ con la condición de que $f - f_s$ sea ortogonal a S o lo que es lo mismo a $\{\nu_1, \nu_2, \cdots, \nu_m\}$, es decir se han de verificar para $j \in \{1, 2, \cdots, m\}$ las condiciones siguientes:

  $$(f - f_s|\nu_j) = 0 \Longleftrightarrow (f|\nu_j) = (f_s|\nu_j)$$

  y sustituyendo λ_s por su expresión anterior, se obtiene al sistema de m

ecuaciones lineales

$$\boxed{\sum_{i=1}^{m} \lambda_i(\nu_i|\nu_j) = (f|\nu_j) \ (j = 1, 2, \cdots, m)}$$

que es compatible determinado (pues su matriz de coeficientes es simétrica y definida positiva), lo que permite obtener las incógnitas $\lambda_1, \cdots, \lambda_m$.

En cambio, si consideramos en $\mathbb{R}^n$ el producto escalar euclídeo, definido $\forall x, y \in \mathbb{R}^n$ por $(x|y) = \sum_{i=1}^{n} x_i y_i$, dado un subespacio vectorial S de dimensión m, de acuerdo con el teorema de la proyección, para todo $y \in \mathbb{R}^n$ existe un único $y_s \in S$ que es **la mejor aproximación por mínimos cuadrados discreta** de y por elementos de S, que se calcula por cualquiera de las dos formas anteriores, teniendo en cuenta que ahora el producto escalar viene dado por una suma.

Veamos a modo de ejemplo el siguiente problema:

- En el espacio euclídeo de las funciones reales continuas definidas en el intervalo $[-\pi, \pi]$, dotado del producto escalar $(f|g) = \int_{-\pi}^{\pi} f(x)g(x)dx$, se pide hallar la mejor aproximación por mínimos cuadrados continua de la función $f(x) = 1 + |x|$ por polinomios trigonométricos de grado menor o igual que dos. Repreentar ambas funciones en ese intervalo.

Nos piden aproximar la función $f(x) = 1 + |x|$ por una función de la forma: $fs(x) = a0 + a1\cos(x) + b1\sin(x) + a2\cos(2x) + b2\sin(2x)$. Para ello hemos de calcular $a0, a1, b1, a2, b2$ tales que sean soluciones del sistema lineal de mínimos cuadrados:

```
(%i1) load(abs_integrate)$ numer:true$
      fs(x):=a0+a1*cos(x)+b1*sin(x)+a2*cos(2*x)+b2*sin(2*x);
      f(x):=1+abs(x);
      eq1:integrate(fs(x),x,-%pi,%pi)=
                      integrate(f(x),x,-%pi,%pi)$
      eq2:integrate(fs(x)*cos(x),x,-%pi,%pi)=
                      integrate(f(x)*cos(x),x,-%pi,%pi)$
      eq3:integrate(fs(x)*sin(x),x,-%pi,%pi)=
                      integrate(f(x)*sin(x),x,-%pi,%pi)$
      eq4:integrate(fs(x)*cos(2*x),x,-%pi,%pi)=
                      integrate(f(x)*cos(2*x),x,-%pi,%pi)$
      eq5:integrate(fs(x)*sin(2*x),x,-%pi,%pi)=
                      integrate(f(x)*sin(2*x),x,-%pi,%pi)$
      linsolve([eq1,eq2,eq3,eq4,eq5],[a0,a1,b1,a2,b2]);
```

(%o3) $\mathrm{fs}\,(x) := a0 + a1\cos(x) + b1\sin(x) + a2\cos(2\,x) + b2\sin(2\,x)$
(%o4) $\mathrm{f}\,(x) := 1 + |x|$
(%o10) $[a0 = 2{,}570796326794899, a1 = -1{,}273239544735164, b1 = 0, a2 = -7{,}7661728742053211\,10^{-16}, b2 = 0]$

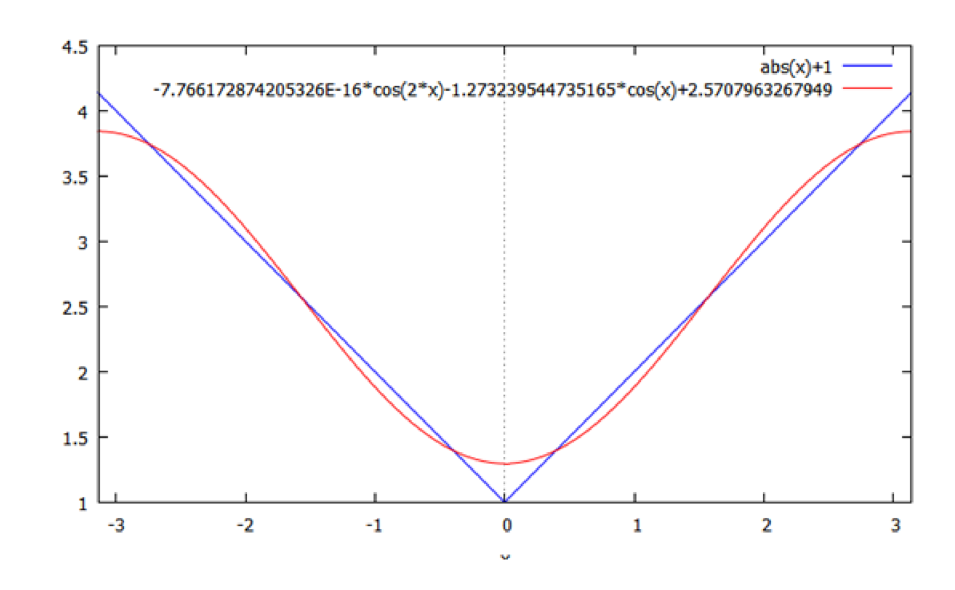
Luego la mejor aproximación pedida será

$\mathrm{fs}\,(x) := 2{,}5707963267949 - 1{,}273239544735165\cos(x) - 7{,}766172874205326\,10^{-16}\cos(2\,x)$

Ahora si representamos ambas en el intervalo $[-\pi, \pi]$, mediante el comando plot2d, podremos apreciar su proximidad.

```
(%i11) kill(all)$ f(x):=1+abs(x)$
       fs(x):=2.5707963267949-1.273239544735165*cos(x)
                  -7.766172874205326*(10^-16)*cos(2*x)$
       plot2d([f(x),fs(x)],[x,-%pi,%pi]);
```

(%o3)



5.5. Ejercicios propuestos

Realizar los ejercicios siguientes utilizando Maxima.

1. Dados los valores aproximados de una función $f(x)$:$f(0{,}0) = 1{,}0$, $f(0{,}5) \simeq 1{,}224745$, $f(1{,}0) \simeq 1{,}414214$, se pide hallar el polinomio de interpolación de dos formas diferentes y utilizarlo para dar un valor aproximado de $f(0{,}4)$.

2. Sea $f(x) = e^x$, calcule el grado mínimo n del polinomio de Taylor $T(x)$ para que se verifique $|f(0{,}1) - T(0{,}1)| < 10^{-5}$.

3. Se considera la siguiente tabla de valores

t	1	3	4
f(t)	-2	0	1

   Se pide:

   - Hallar el spline cúbico natural que ajusta estos datos.
   - Utilizar el spline para estimar $f(2)$, $f'(3)$ y $f''(3)$.

4. ¿En cuántos subintervalos iguales es necesario dividir el intervalo de integración para aproximar por la regla del trapecio compuesta, con error menor que $5 \cdot 10^{-7}$, la integral $\int_0^1 \frac{dx}{x^2+1}$?. Hallar dicha aproximación y comparar con la solución exacta.

5. Aproximar por la regla de Simpson compuesta, con error menor que 10^{-7}, la integral $\int_0^1 \sqrt{1+x}dx$?. Comparar con la solución exacta.

6. Aproximar por la regla de Simpson compuesta, con error menor que 10^{-7}, la integral $\int_0^1 xcosxdx$.

Capítulo 6

Métodos numéricos de integración de PVI para EDOs

6.1. Método de Euler

El método de Euler es el método numérico más simple para integrar numéricamente un PVI: $y' = f(t, y)$, $y(t_0) = y_0$, consiste en generar las aproximaciones y_n, a la solución en t_n, en la forma $y_{n+1} = y_n + h \cdot f(t_n, y_n)$.

Comenzaremos aplicándolo al PVI: $y' = y, y(0) = 1$; que integraremos en el intervalo $[0, 1]$, tomando $h = 0{,}1$, obteniendo los valores aproximados de la solución en los puntos de la red y representando la solución aproximada frente a la exacta, dada en este caso por $y(t) = e^t$. Un programa sencillo de Maxima para aplicarlo en este caso es el siguiente.

```
(%i1)  kill(all)$ numer:true$
       T:1$ N:10$ h:T/N$ t[0]:0$ define(f(t,y),y)$ y[0]:1$
       for k:1 thru 10 do
       (t[k]:t[0]+k*h,y[k]:y[k-1]+h*f(t[k-1],y[k-1]))$
       print("Salida numérica: ",sol:makelist([t[j],y[j]],j,0,N))$
       print("Solución exacta versus solución aproximada:")$
       wxplot2d([[discrete,sol],%e^t],[t,0,1],[style,[points,2,2],
       [lines,1,1]],[legend,"y(t) aproximada","y(t) exacta"],
       [xlabel,"t"], [ylabel,"y"]);
```

Salida numérica:
[[0,1],[0.1,1.1],[0.2,1.21],[0.3,1.331],[0.4,1.4641],[0.5,1.61051],[0.6,1.771561],
[0.7,1.9487171],[0.8,2.14358881],[0.9,2.357947691],[1.0,2.5937424601]]

El método de Euler nos ha dado los valores aproximados de la solución en los puntos: 0, 0.1, 0.2,..., 0.9 y 1.0. En la gráfica que sigue comparamos la

solución numérica con la exacta.

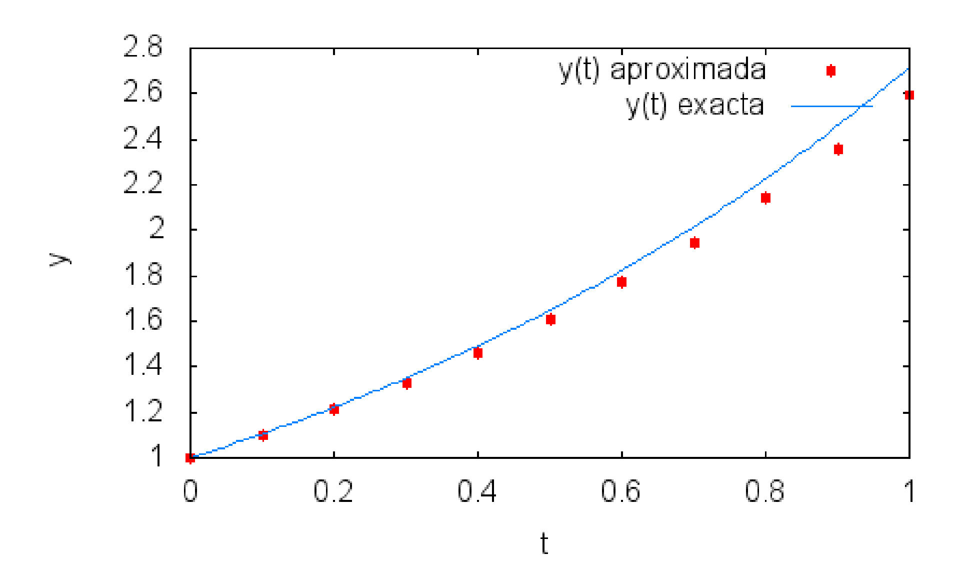


Figura 6.1: Solución exacta versus aproximada con h = 0.1

Si se quiere aplicar el método a cualquier otro PVI, basta cambiar la definición de f, la condición inicial es decir el t_0 y el y_0, la amplitud T del intervalo de integración y el número N de subintervalos en que se divide, que está relacionado con h, pues es $h = T/N$. En general, no se conoce la solución exacta por lo que se puede suprimir la comparación entre las gráficas, que aquí hemos incluido. Asimismo, normalmente no imprimiremos toda la solución numérica hallada sino únicamente los valores que nos pidan, por ejemplo si quiero sólo los valores que nos da el método anterior en los puntos $t = 0{,}5$ y $t = 1$, podemos hacerlo como sigue:

```
(%i12) kill(all)$ numer:true$
       T:1$ N:10$ h:T/N$ t[0]:0$ define(f(t,y),y)$ y[0]:1$
       for k:1 thru 10 do
       (t[k]:t[0]+k*h,y[k]:y[k-1]+h*f(t[k-1],y[k-1]))$
       sol:makelist([t[j],y[j]],j,0,N)$ print( "y(0.5) ~ ",y[5],
       "    y(1.0) ~ ",y[10])$
```

Cuya salida es $y(0{,}5) \simeq 1{,}61051$ e $y(1{,}0) \simeq 2{,}5937424601$.

Como sabemos se trata de un método numérico convergente de orden uno, ahora integramos el mismo problema tomando $h = 0{,}01$, para ello basta cambiar en el primer programa el valor de N, tomando ahora $N = 100$.

```
(%i12) kill(all)$ numer:true$
       T:1$ N:100$ h:T/N$ t[0]:0$
       kill(f)$ define(f(t,y),y)$ y[0]:1$
       for k:1 thru N do
       (t[k]:t[0]+k*h,y[k]:y[k-1]+h*f(t[k-1],y[k-1]))$
       print("Salida numérica: ",sol:makelist([t[j],y[j]],j,0,N))$
       print("Solución exacta versus solución aproximada:")$
       wxplot2d([[discrete,sol],%e^t],[t,0,1],[style,[points,2,2],
       [lines,1,1]],[legend,"y(t) aprox","y(t) real"],[xlabel,"t"],
       [ylabel,"y"]);
```

Salida numérica: [[0,1],[0.01,1.01],[0.02,1.0201],[0.03,1.030301],[0.04,1.04060401], [0.05,1.0510100501], [0.06,1.061520150601], [0.07,1.07213535210701],......, [0.98,2.651518311363127],[0.99,2.678033494476758],[1.0,2.704813829421526]]

El método de Euler nos da los valores aproximados de la solución en los puntos: 0, 0.01, 0.02,..., 0.99 y 1.0 de la red en [0,1], definidos por $h = 0{,}01$.

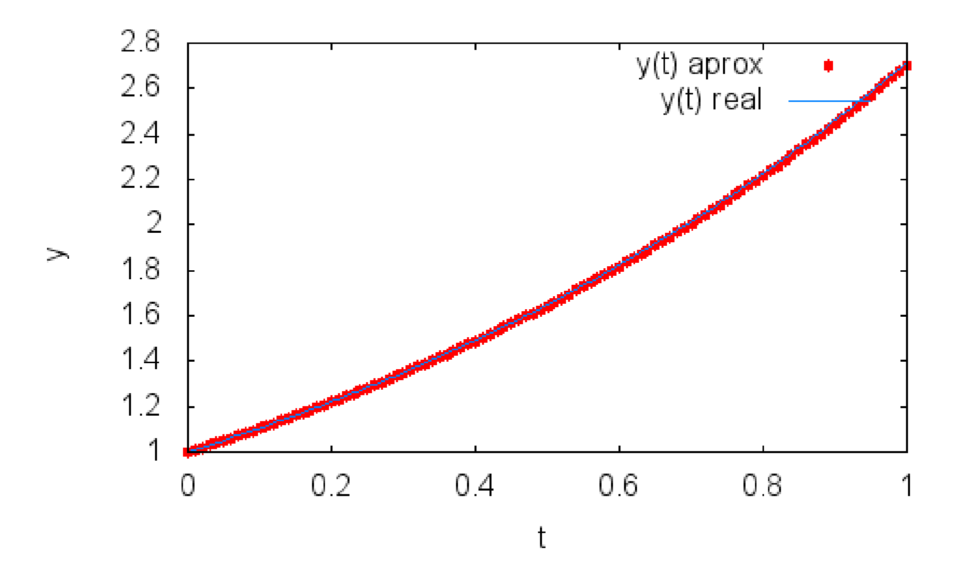


Figura 6.2: Solución exacta solución aproximada con h = 0.01

Si nuevamente sólo queremos los valores aproximados en 0,5 y 1,0, es suficiente con hacer lo siguiente

```
(%i11) kill(all)$ numer:true$
       T:1$ N:100$ h:T/N$ t[0]:0$ define(f(t,y),y)$ y[0]:1$
       for k:1 thru N do
       (t[k]:t[0]+k*h,y[k]:y[k-1]+h*f(t[k-1],y[k-1]))$
       sol:makelist([t[j],y[j]],j,0,N)$ print( "y(0.5) ~ ",y[50],
       "     y(1.0) \simeq ",y[100])$
```

Que nos da $y(0{,}5) \simeq 1{,}6446318218438882$ e $y(1{,}0) \simeq 2{,}704813829421526$.

Aunque en la última gráfica parecen coincidir la solución aproximada y la exacta, el caso es que el valor aproximado que se obtiene al final del intervalo resulta ser $y_{100} = 2{,}704813829421526$ mientras que el exacto es $y(1{,}0) = e = 2{,}718281828459045$, no obstante hay mucha diferencia para la gran cantidad de cálculos realizada. Podemos ir haciendo $N = 1000, 10000, ...$, pero el tiempo de cálculo y los resultados obtenidos hacen dudar de su eficiencia.

Veamos el siguiente programa para el método de Euler, dado por un block, que denominamos "Eulerpuntofinal", en el que le pedimos sólo el valor aproximado de la solución del PVI:

$$y' = f(t, y), \;\; y(t_0) = y_0$$

al final del intervalo de integración, es decir en el punto $t_0 + T$. Lo aplicaremos al anterior problema de valor inicial $y' = y, y(0) = 1$ para hallar el valor aproximado de la solución en el punto 1,0 y compararla con el valor exacto, mostrando también el tiempo de cálculo empleado, que muestra la poca eficiencia computacional del método de Euler. Para mostrar el tiempo de cálculo utilizamos el comando "showtime:true". Como sabemos, por defecto esta variable está en false, pero una vez que la declaramos como true sigue así hasta que reiniciemos o bien la volvamos a declarar como false mediante el comando "showtime:false".

```
(%i11) Eulerpuntofinal(N,T,t,y,f):=block(numer:true,
       h:T/N,for i:1 thru N do (y:y+h*f(t,y),t:t+h),
       print("y(",t,") es aproximado por ", y)
       )$
```

En este programa N es el número de partes iguales en que se subdivide el intervalo de integración, T es la amplitud de dicho intervalo, el t de entrada es el punto inicial en el que se da el valor inicial y, en tanto que f es la función que describe la ecuación diferencial de primer orden a integrar. Veamos algunos cálculos con el problema anterior mostrando el tiempo empleado:

```
(%i12) showtime:true$
```

Salida: Evaluation took 0.0000 seconds (0.0000 elapsed)

```
(%i13) f(t,y):=y$ Eulerpuntofinal(10,1,0,1,f)$
```

Salida: Evaluation took 0.0000 seconds (0.0000 elapsed)
y(1,0) es aproximado por 2,5937424601
Evaluation took 0.0000 seconds (0.0000 elapsed)

```
(%i15) Eulerpuntofinal(100,1,0,1,f)$
```

Salida: y(1,000000000000001) es aproximado por 2,704813829421526
Evaluation took 0.0100 seconds (0.0100 elapsed)

```
(%i16) Eulerpuntofinal(1000,1,0,1,f)$
```

Salida: y(1,000000000000001) es aproximado por 2,716923932235896
Evaluation took 0.0000 seconds (0.0000 elapsed)

```
(%i17) Eulerpuntofinal(10000,1,0,1,f)$
```

Salida: y(0,99999999999991) es aproximado por 2,718145926825227
Evaluation took 0.0700 seconds (0.0700 elapsed)

```
(%i18) Eulerpuntofinal(10^5,1,0,1,f)$
```

Salida: y(0,99999999999808) es aproximado por 2,718268237174493
Evaluation took 0.4900 seconds (0.4900 elapsed)

```
(%i19) Eulerpuntofinal(10^6,1,0,1,f)$
```

Salida: y(1,000000000007918) es aproximado por 2,718280469319462
Evaluation took 4.4000 seconds (4.4000 elapsed)

```
(%i20) Eulerpuntofinal(10^7,1,0,1,f)$
```

Salida: y(0,99999999975017) es aproximado por 2,718281692544055
Evaluation took 42.8000 seconds (42.8000 elapsed)

```
(%i21) showtime:false$
```

Es fácil ver que en el caso particular que nos ocupa, la aproximación de $y(1{,}0)$ está dada por $y_N = (1+h)^N$ con $h = 1/N$, pues es $f(t,y) = y$. Como se observa en los casos que siguen, la solución aproximada en el punto 1,0, cuando h va disminuyendo va mejorando pero llega un momento a partir del cual vuelve a empeorar debido a la acumulación de errores de redondeo. (¡Ojo!, da la solución numérica rápidamente en numer:true). Este

comportamiento es habitual en todos los métodos de un paso.

```
(%i22) N:10^3$ h:1/10^3$ print("la solución aproximada con
       10^3 pasos es ",(1+h)^(10^3)," y el error global es ",
       %e-(1+h)^(10^3))$
```

Salida: la solución aproximada con 10^3 pasos es 2,716923932235521 y el error global es 0,0013578962235243

```
(%i25) N:10^4$ h:1/10^4$ print("la solución aproximada con
       10^4 pasos es ",(1+h)^(10^4)," y el error global es ",
       %e-(1+h)^(10^4))$
```

Salida: la solución aproximada con 10^4 pasos es 2,718145926824356 y el error global es $1{,}3590163468890637 \cdot 10^{-4}$

```
(%i28) N:10^5$ h:1/10^5$ print("la solución aproximada con
       10^5 pasos es ",(1+h)^(10^5)," y el error global es ",
       %e-(1+h)^(10^5))$
```

Salida: la solución aproximada con 10^5 pasos es 2,718268237197528 y el error global es $1{,}3591261516676667 \cdot 10^{-5}$

```
(%i31) N:10^6$ h:1/10^6$ print("la solución aproximada con
       10^6 pasos es ",(1+h)^(10^6)," y el error global es ",
       %e-(1+h)^(10^6))$
```

Salida: la solución aproximada con 10^6 pasos es 2,718280469156428 y el error global es $1{,}3593026175762191 \cdot 10^{-6}$

```
(%i34) N:10^7$ h:1/10^7$ print("la solución aproximada con
       10^7 pasos es ",(1+h)^(10^7)," y el error global es ",
       %e-(1+h)^(10^7))$
```

Salida: la solución aproximada con 10^7 pasos es 2,718281693980372 y el error global es $1{,}344786726420466 \cdot 10^{-7}$

```
(%i37) N:10^8$ h:1/10^8$ print("la solución aproximada con
       10^8 pasos es ",(1+h)^(10^8)," y el error global es ",
       %e-(1+h)^(10^8))$
```

Salida: la solución aproximada con 10^8 pasos es 2,718281786395798 y el error global es $4{,}2063247551737959 \cdot 10^{-8}$

Observar que en este último cálculo ha cambiado la constante del error debido al aumento de los errores de redondeo y como empeora el error global

en los siguientes, por esta causa, a medida que h sigue decreciendo o lo que es lo mismo, aumentando el número de subintervalos.

```
(%i40) N:10^9$ h:1/10^9$ print("la solución aproximada con
       10^9 pasos es ",(1+h)^(10^9)," y el error global es ",
       %e-(1+h)^(10^9))$
```

Salida: la solución aproximada con 10^9 pasos es 2,71828203081451 y el error global es $-2{,}023554643848513 \cdot 10^{-7}$

```
(%i43) N:10^12$ h:1/10^12$ print("la solución aproximada con
       10^12 pasos es ",(1+h)^(10^12)," y el error global es ",
       %e-(1+h)^(10^12))$
```

Salida: la solución aproximada con 10^{12} pasos es 2,71852346956828 y el error global es $-2{,}4164110923452498 \cdot 10^{-4}$

```
(%i46) N:10^15$ h:1/10^15$ print("la solución aproximada con
       10^15 pasos es ",(1+h)^(10^15)," y el error global es ",
       %e-(1+h)^(10^15))$
```

Salida: la solución aproximada con 10^{15} pasos es 3,035035181396292 y el error global es $-0{,}31675335293725$

Observación. En general, para métodos de un paso y orden 1 se recomienda **no tomar pasos de longitud menor que la raíz cuadrada del épsilon de máquina**, es decir menores que $\sqrt{2^{-52}} = 1{,}4901161193847656 \cdot 10^{-8}$. Si el orden del método de un paso fuera p se recomienda **no tomar pasos inferiores a la raíz de índice** $(p+1)$ **del épsilon de máquina**, es decir a $\sqrt[p+1]{2^{-52}}$.

```
(%i49) %e,numer;sqrt(2^(-52));
```

$(\%o49)$ 2,718281828459045
$(\%o50)$ $1{,}4901161193847656\, 10^{-8}$

Seguiremos viendo algunas variantes del programa anterior aplicadas al ejemplo del PVI: $y' = y, y(0) = 1$ en el intervalo $[0, 1]$. Con pequeños cambios los programas de este párrafo y los siguientes se adaptan a otros PVI así como a una ecuación de orden superior al primero o a un sistema de ecuaciones diferenciales. Si bien en general, como no se conoce la solución exacta, no podremos hacer la comparación ni el dibujo de esta. El siguiente programa es una pequeña variación del de más arriba, en el que se realizan y guardan los $t[n]$ e $y[n]$, pero sólo se muestra el último valor calculado $y[N]$:

```
(%i51) Eulerredpuntofinal(N,T):=block(
       numer:true,kill(y,t,f),h:T/N,t[0]:0,
       y[0]:1,define(f(t,y),y),
       for n:1 thru N do (t[n]:t[0]+n*h,
       y[n]:y[n-1]+h*f(t[n-1],y[n-1])),
       display(y[N])
       )$
```

```
(%i52) Eulerredpuntofinal(100,1)$
```

$y_{100} = 2{,}704813829421526$

```
(%i53) Eulerredpuntofinal(200,1)$
```

$y_{200} = 2{,}711517122929374$

```
(%i54) Eulerredpuntofinal(400,1)$
```

$y_{400} = 2{,}714891744381287$

```
(%i55) Eulerredpuntofinal(800,1)$
```

$y_{800} = 2{,}716584846682537$

En tanto que en el siguiente se se guardan los $[t[n], y[n]]$ en una lista y se muestra esta, siempre que no sea excesivamente larga.

```
(%i56) Eulertodalared(N,T):=block(
       numer:true,kill(y,t,f),
       h:T/N,t[0]:0,y[0]:1,define(f(t,y),y),
       for n:1 thru N do (t[n]:t[0]+n*h,
       y[n]:y[n-1]+h*f(t[n-1],y[n-1])),
       sol:makelist([t[j],y[j]],j,0,N)
       )$
```

Si ahora ponemos por ejemplo: Eulertodalared(100,1); Eulertodalared(200,1); Eulertodalared(400,1); Eulertodalared(800,1); vemos que hace y presenta todos los resultados para los tres primeros pero no así para el último, que nos da como salida el mensaje que sigue:

```
(%i60) Eulertodalared(800,1);
```

« ¡Expresión excesivamente larga para ser mostrada! », en este caso la hace pero no la muestra por ser excesivamente larga, como se pone de manifiesto en el programa que sigue, en el que le pedimos además que dibuje la gráfica

resultante, comparándola con la de la solución exacta y mostrando el tiempo empleado, mediante la instrucción:

```
(%i61) showtime:true$
```

Cuya salida es “Evaluation took 0.0000 seconds (0.0000 elapsed)”.

En tanto que para las gráficas de las soluciones aproximada y exacta podemos utilizar el siguiente programa:

```
(%i62) Eulergrafico(N,T):=block(numer:true,
       kill(y,t,f),h:T/N,t[0]:0,y[0]:1,
       define(f(t,y),y),
       for k:1 thru N do (t[k]:t[0]+k*h,
       y[k]:y[k-1]+h*f(t[k-1],y[k-1])),
       sol:makelist([t[j],y[j]],j,0,N),
       wxplot2d([[discrete,sol],%e^t],[t,0,1],[y,1,%e],
       [style,[points,2,2],[lines,1,1]],[legend,"y(t)
       aproximada","y(t) exacta"],[xlabel,"t"],
       [ylabel,"y"]))$
```

Que ejecutaremos para N= 100, 200, 400,..., 51200 dándonos las gráficas correspondientes y los tiempos de cálculo empleados:

```
(%i63) Eulergrafico(10,1)$
```

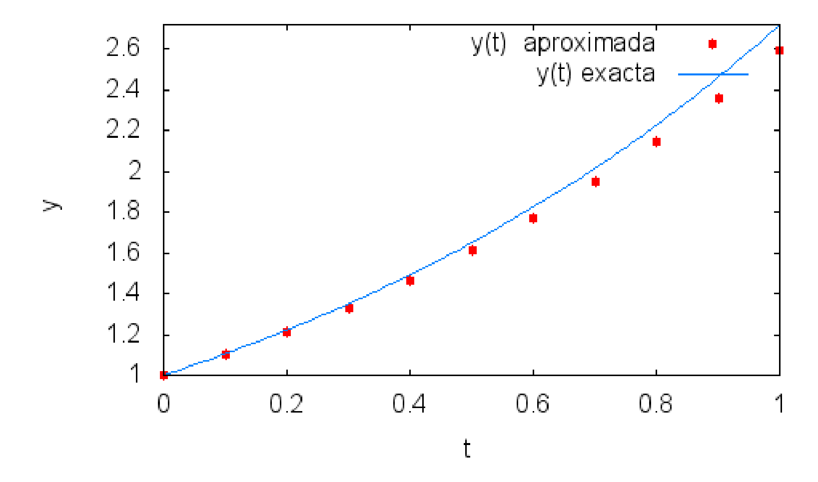


Figura 6.3: Solución exacta versus aproximada con N = 10

«Evaluation took 0.1000 seconds (0.1000 elapsed)»

```
(%i64) Eulergrafico(100,1)$
```

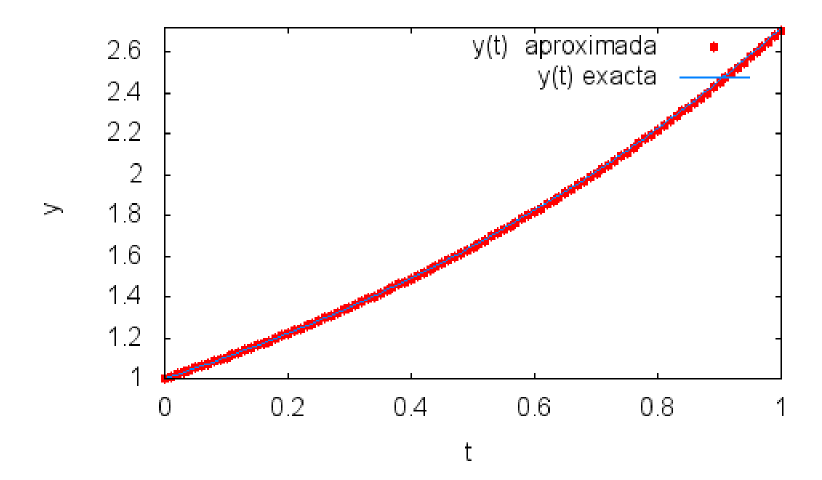


Figura 6.4: Solución exacta versus aproximada con N = 100

«Evaluation took 0.1200 seconds (0.1200 elapsed)»

```
(%i65) Eulergrafico(200,1);
```

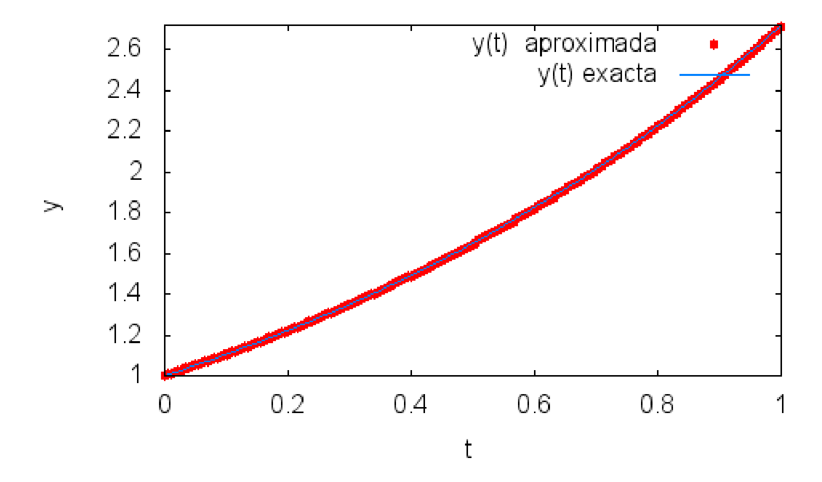


Figura 6.5: Solución exacta versus aproximada con N = 200

«Evaluation took 0.1300 seconds (0.1300 elapsed)»

```
(%i66) Eulergrafico(400,1);
```

Que da la salida

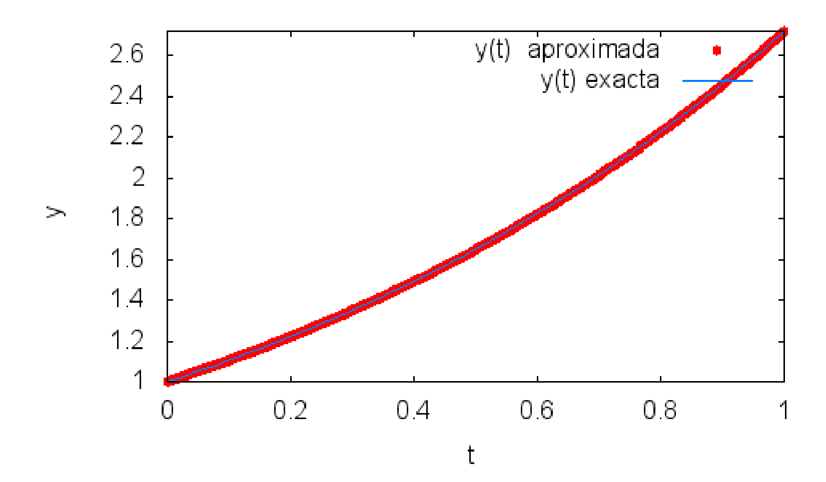


Figura 6.6: Solución exacta versus aproximada con N = 400

«Evaluation took 0.1400 seconds (0.1400 elapsed)»

```
(%i67) Eulergrafico(800,1);
```

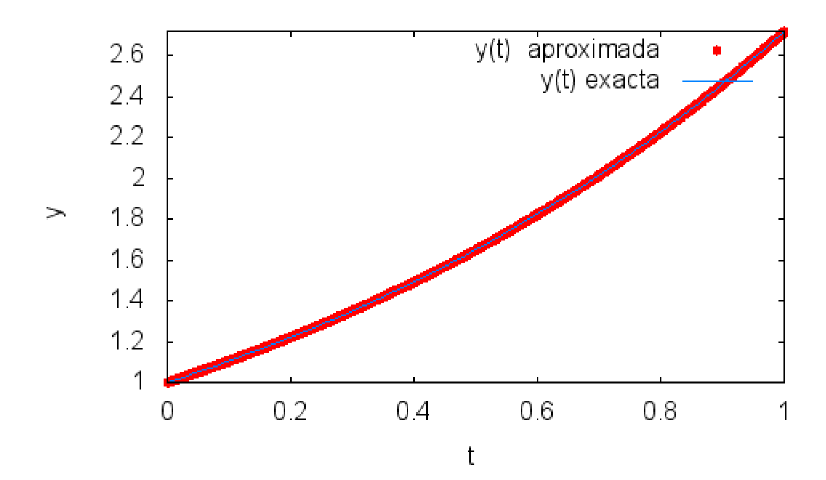


Figura 6.7: Solución exacta versus aproximada con N = 800

«Evaluation took 0.1600 seconds (0.1600 elapsed)»

```
(%i68) Eulergrafico(1600,1);
```

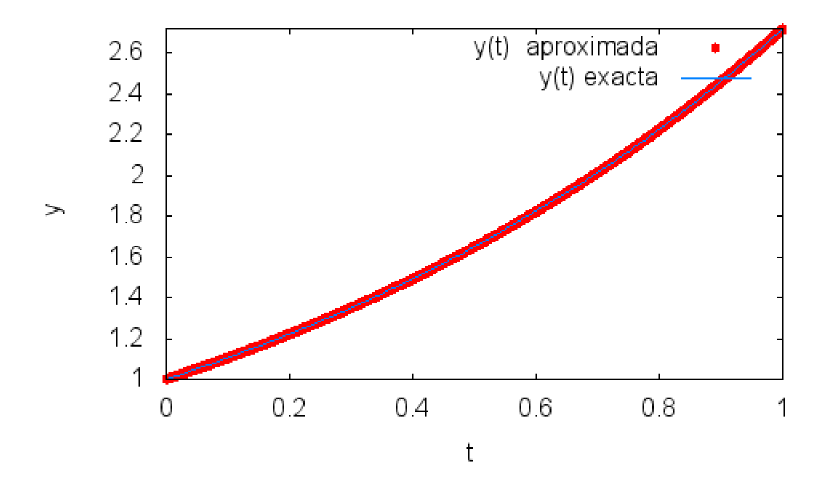


Figura 6.8: Solución exacta versus aproximada con N = 1600

«Evaluation took 0.2100 seconds (0.2100 elapsed)»

```
(%i69) Eulergrafico(3200,1);
```

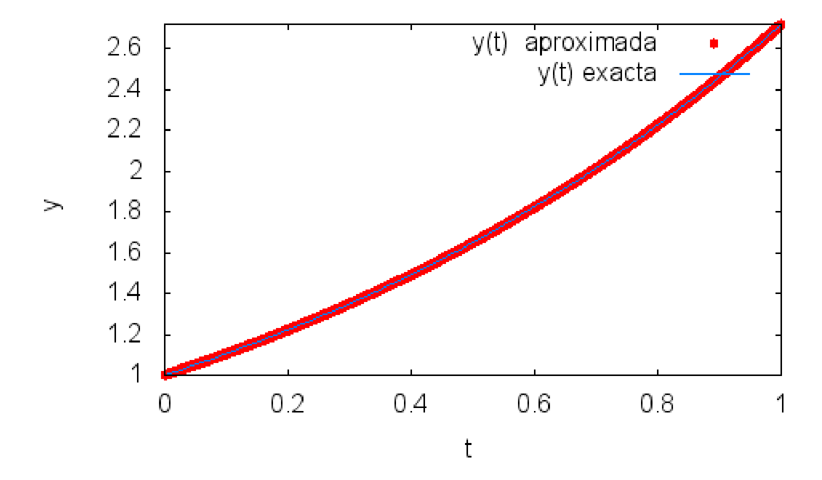


Figura 6.9: Solución exacta versus aproximada con N = 3200

«Evaluation took 0.2800 seconds (0.2800 elapsed)»

```
(%i70) Eulergrafico(6400,1);
```

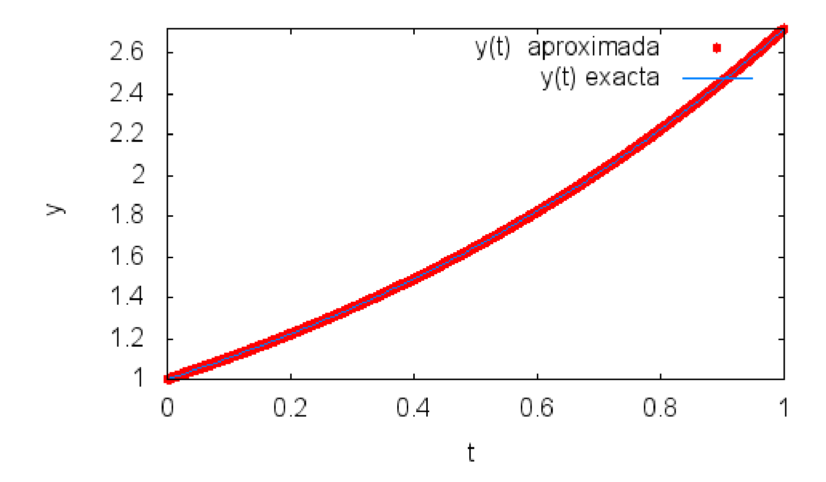


Figura 6.10: Solución exacta versus aproximada con N = 6400

«Evaluation took 0.4700 seconds (0.4700 elapsed)»

```
(%i71) Eulergrafico(12800,1);
```

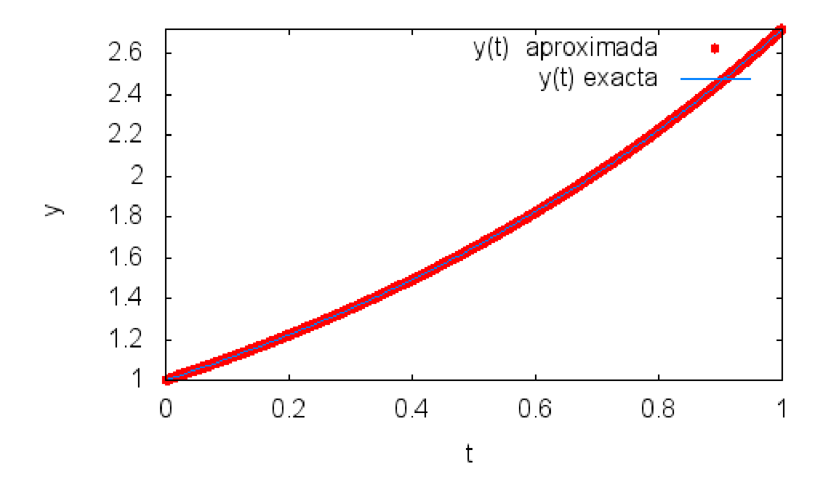


Figura 6.11: Solución exacta versus aproximada con N = 12800

«Evaluation took 0.8200 seconds (0.8200 elapsed)»

```
(%i72) Eulergrafico(25600,1);
```

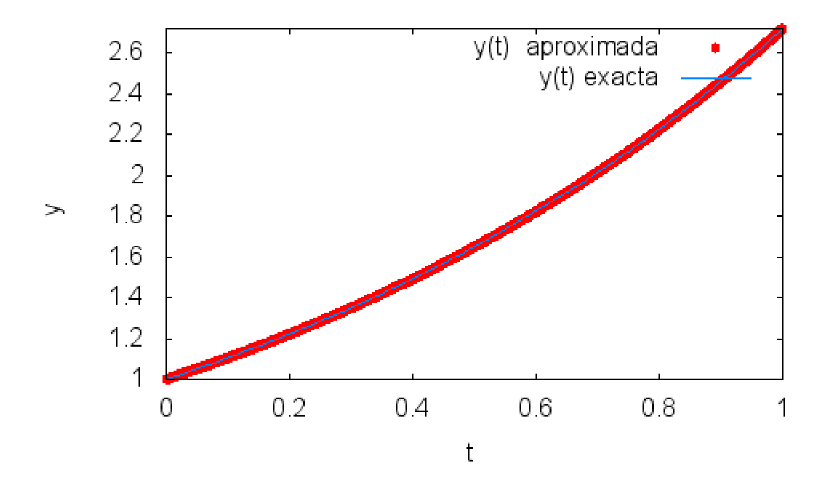


Figura 6.12: Solución exacta versus aproximada con N = 25600

«Evaluation took 1.5800 seconds (1.5800 elapsed)»

```
(%i73) Eulergrafico(51200,1);
```

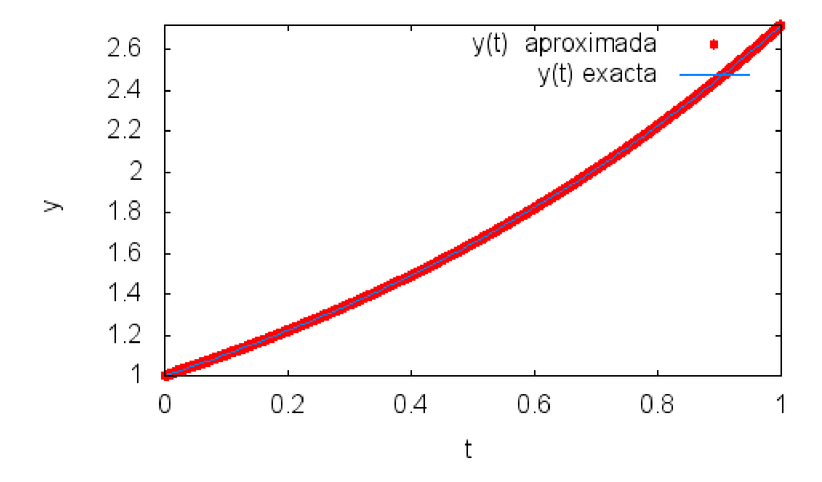


Figura 6.13: Solución exacta versus aproximada con N = 51200

«Evaluation took 3.0000 seconds (3.0000 elapsed)»

```
(%i74) showtime:false;
```

(%o74) *false*

Notas: 1) Con pequeños cambios los programas de este párrafo y los siguientes se adaptan a otros PVI, así como a una ecuación de orden superior al primero o a un sistema de ecuaciones diferenciales. Si bien en general no se conoce la solución exacta y no podríamos hacer la comparación gráfica ni numérica.

2) Se observa que, en aritmética de redondeo, a medida que N crece y se va duplicando el tiempo empleado aproximadamente también se duplica.

6.2. Métodos de Taylor

Los métodos de Taylor son métodos de un paso de orden superior al primero, que se obtienen truncando el desarrollo de Taylor por el orden pedido, a modo de ejemplo vamos a integrar numéricamente el anterior problema de valor inicial: $y' = y, y(0) = 1$ en el intervalo $[0, 1]$, con un método de Taylor de orden 3, tomando h = 0.1. Luego, comparamos gráficamente la solución discreta obtenida con la exacta, dada en este caso por $y = e^t$ y mostramos el error global en el punto $t = 1{,}0$.

```
(%i76) kill(all)$ numer:true$
       N:10$ h:1/N$ y[0]:1$ t[0]:0$
       kill(f,f1,f2,fi)$
       define(f(t,y),y);
       define(f1(t,y),diff(f(t,y),t)+diff(f(t,y),y)*f(t,y));
       define(f2(t,y),(diff(f1(t,y),t)+diff(f1(t,y),y)*f(t,y)));
       define(fi(t,y),f(t,y)+(h/2)*f1(t,y)+(h^2/6)*f2(t,y));
       for k:1 thru N do (
       t[k]:t[0]+k*h,y[k]:y[k-1]+h*fi(t[k-1],y[k-1]))$
       print("Solución numérica: ",sol:makelist([t[j],y[j]],j,0,N))$
       wxplot2d([[discrete,sol],%e^t],[t,0,1],[style,[points,2,2],
       [lines,1,1]],[legend,"y(t) aproximada","y(t) exacta"],
       [xlabel,"t"],[ylabel,"y"]);
       print("El error global en 1.0 es EG(1.0) = ",%e-y[N])$
```

(%o7) $f(t, y) := y$
(%o8) $f1(t, y) := y$
(%o9) $f2(t, y) := y$
(%o10) $fi(t, y) := 1{,}051666666666667y$

Solución numérica: [[0,1],[0.1,1.105166666666667],[0.2,1.221393361111111],[0.3,1.349843229587963],[0.4,1.491801742566297],[0.5,1.648689559159519],[0.6,1.822076744464462],[0.7,2.013698482090641],[0.8,2.22547243912384],[0.9,2.459517957305031],[1.0,2.71817726248161]]

(%t13)

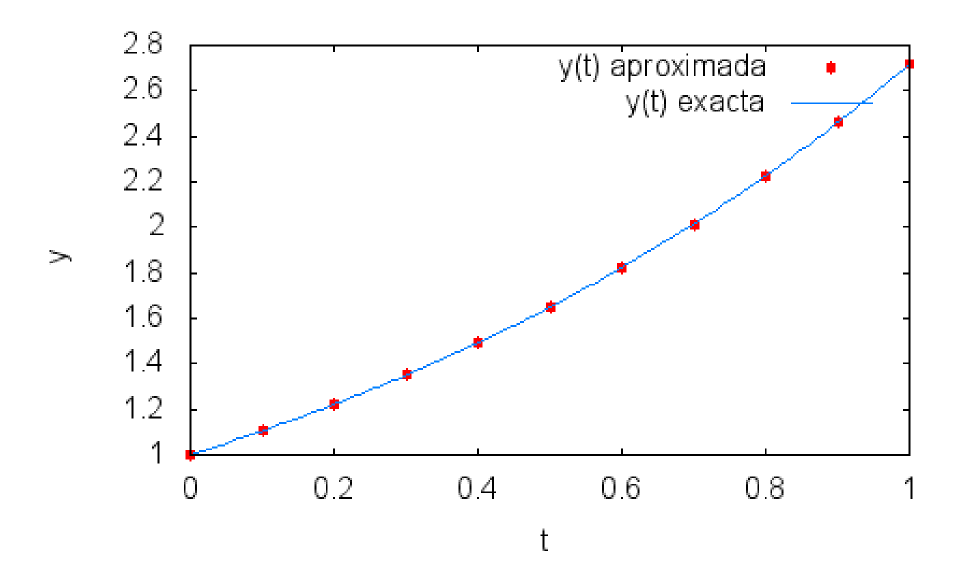


Figura 6.14: Exacta vs aproximada por Taylor de orden 3 y h = 0.1

(%o13)
El error global en 1,0 es $EG(1{,}0) = 1{,}0456597743546681 \cdot 10^{-4}$

Observemos que con $N = 10$ se obtiene una mejor aproximación que el método de Euler con $N = 10000$, de ahí el interés de tener convergencia de orden elevado, se obtienen buenas aproximaciones con menos cálculos y por tanto con menos errores de redondeo y más eficiencia computacional. Además, de este modo podemos obtener métodos del orden que deseemos, con el único inconveniente de tener que realizar y evaluar las derivadas sucesivas de la función que define la ecuación diferencial. Si el orden es tres, el error global es el producto de una constante C (independiente de h) por h^3, se puede constatar que si se divide h por 10 el error global se divide aproximadamente por 10^3. Lo vemos a continuación aplicando el método de Taylor de orden 3 anterior con $N = 100$.

```
(%i15) kill(all)$ numer:true$
       N:100$ h:1/N$ y[0]:1$ t[0]:0$
       kill(f,f1,f2,fi)$
       define(f(t,y),y);
       define(f1(t,y),diff(f(t,y),t)+diff(f(t,y),y)*f(t,y));
       define(f2(t,y),(diff(f1(t,y),t)+diff(f1(t,y),y)*f(t,y)));
       define(fi(t,y),f(t,y)+(h/2)*f1(t,y)+(h^2/6)*f2(t,y));
       for k:1 thru N do (
       t[k]:t[0]+k*h,y[k]:y[k-1]+h*fi(t[k-1],y[k-1]))$
       print("Solución numérica: ",sol:makelist([t[j],y[j]],j,0,N))$
       wxplot2d([[discrete,sol],%e^t],[t,0,1],[style,[points,2,2],
       [lines,1,1]],[legend,"y(t) aproximada","y(t) exacta"],
       [xlabel,"t"],[ylabel,"y"]);
       print("El error global en 1.0 es EG(1.0) = ",%e-y[N])$
```

(%o7) $f(t,y) := y$

(%o8) $f1(t,y) := y$

(%o9) $f2(t,y) := y$

(%o10) $fi(t,y) := 1{,}005016666666667y$

Solución numérica: la omitimos para abreviar.

(%t13)

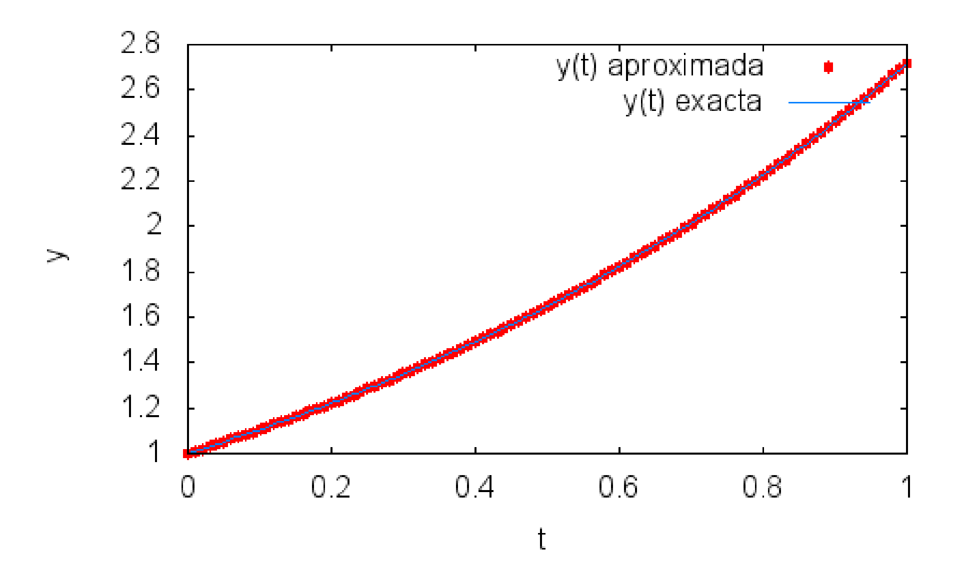


Figura 6.15: Exacta vs aproximada por Taylor de orden 3 y h = 0.01

(%o13) El error global en 1,0 es $EG(1{,}0) = 1{,}1235941110854242 \cdot 10^{-7}$

Observación. Se observa que el error global funciona como era de esperar, pues al dividir h por 10 el error global se ha dividido aproximadamente por 10^3, concretamente por $930{,}638\cdots$.

Si se desea se puede escribir este programa en un block y aplicarlo a ecuaciones diferentes, con ligeros cambios sobre el anterior. Veamos un ejemplo para el método de Taylor de orden 3 para nuestro problema:

```
(%i16) Taylor3(N,T):=block(kill(y,f,f1,f2,fi,t),N,T,
       numer:true,h:1/N, y[0]:1, t[0]:0,
       define(f(t,y),y),
       define(f1(t,y),diff(f(t,y),t)+diff(f(t,y),y)*f(t,y)),
       define(f2(t,y),(diff(f1(t,y),t)+diff(f1(t,y),y)*f(t,y))),
       define(fi(t,y),f(t,y)+(h/2)*f1(t,y)+(h^2/6)*f2(t,y)),
       for k:1 thru N do (
       t[k]:t[0]+k*h,y[k]:y[k-1]+h*fi(t[k-1],y[k-1])),
       print("Solución numérica: ",sol:makelist([t[j],y[j]],j,0,N)),
       wxplot2d([[discrete,sol],%e^t],[t,0,1],[style,[points,2,2],
       [lines,1,1]],[legend,"y(t) aproximada","y(t) exacta"],
       [xlabel,"t"],[ylabel,"y"]),
       print("El error global en 1.0 es EG(1.0) = ",%e-y[N]))$
```

Si ahora tecleamos “Taylor3(10,1);” o bien “Taylor3(100,1);” obtenemos las salidas anteriores.

Ejercicio. Escribir un programa, mediante un block, para integrar numéricamente el PVI anterior con un método de Taylor de orden 4.

6.3. Métodos Runge-Kutta para EDOs

6.3.1. Métodos RK(3) explícitos

Para los métodos Runge-Kutta (abreviadamente RK) explícitos, que son los únicos que abordaremos de momento, el programa constará, en general, de varios bloques, en el primero de ellos hemos de definir el PVI a integrar, es decir los valores iniciales, la longitud del intervalo de integración y la ecuación diferencial, en el segundo hemos de introducir el tablero de Butcher correspondiente al método Runge-Kutta explícito de tres etapas que vamos a utilizar, luego debemos programar el esquema del método y finalmente la salida deseada, que puede ser el valor aproximado de la solución en el punto final del intervalo o los valores aproximados en toda la red de puntos consi-

derada y/o su representación gráfica. También podemos determinar antes de comenzar el orden del método mediante las ecuaciones de orden.

En cada paso el esquema es siempre el mismo para cualquier método Runge- Kutta: se calculan los valores auxiliares ki (en este caso $k1$, $k2$ y $k3$) y a partir de ellos se actualiza la solución y_n, también hay que actualizar el valor temporal $t_n = t_{n-1} + h$. Para evitar que la memoria se desborde se suelen sobrescribir tanto t_n como y_n, sobre todo si sólo deseamos presentar el valor en el punto final del intervalo, aunque si el número N no es muy grande se pueden guardar todos y hacer su gráfica como antes. Recordemos que cada $ki(i = 1, 2, 3)$ es de la forma:

$k[i] : f(t + c[i] \cdot h, y + h \cdot sum(a[i, j] \cdot k[j], j, 1, i - 1))$

En tanto que

$y[n] : y[n - 1] + h \cdot sum(b[i] \cdot k[i], i, 1, 3)$

Por otro lado, recordemos que las ecuaciones de orden para los métodos RK explícitos de tres etapas, supuesto que se cumplen las condiciones de simplificación, son:

1. $b[1] + b[2] + b[3] = 1$

2. $b[2] \cdot c[2] + b[3] \cdot c[3] = 1/2$

3. $b[2] \cdot c[2]^2 + b[3] \cdot c[3]^2 = 1/3$ (3.1)
   y $b[3] \cdot a[3, 2] \cdot c[2] = 1/6$ (3.2)

De manera que si se verifica la 1) pero no la 2) el método es sólo de primer orden, si se verifica la 1) y la 2) pero no se verifica alguna de las (3.1) o (3.2) es de segundo orden y si se verifican todas es de tercer orden (el máximo alcanzable por un método Runge-Kutta explícito de tres etapas). Para dar un método RK(3) explícito hay que dar los coeficientes $b[i](i = 1, 2, 3)$, los $c[2]$, $c[3]$, así como los $a[2, 1]$, $a[3, 1]$ y $a[3, 2]$. Además, podemos comprobar su orden mediante las ecuaciones anteriores. Vamos ahora a integrar nuevamente el problema text anterior con el método RK explícito de tres etapas para el que

$$b[1] = 1/6,\ b[2] = 4/6,\ b[3] = 1/6,\ c[2] = 1/2,\ c[3] = 1$$

$$a[2, 1] = 1/2,\quad a[3, 1] = -1,\quad a[3, 2] = 2$$

antes de nada, puesto que:

```
(%i19) numer:false$ b[1]:1/6$ b[2]:4/6$ b[3]:1/6$ c[2]:1/2$ c[3]:1$
       a[2,1]:1/2$ a[3,1]:-1$ a[3,2]:2$
       print("b[1]+b[2]+b[3]  = ",b[1]+b[2]+b[3])$
       print("b[2]*c[2]+b[3]*c[3]  = ",b[2]*c[2]+b[3]*c[3])$
       print("b[2]*c[2]^2+b[3]*c[3]^2 = ",b[2]*c[2]^2+b[3]*c[3]^2)$
       print("b[3]*a[3,2]*c[2]  = ", b[3]*a[3,2]*c[2])$
```

$b[1] + b[2] + b[3] = 1$
$b[2] * c[2] + b[3] * c[3] = \frac{1}{2}$
$b[2] * c[2]^2 + b[3] * c[3]^2 = \frac{1}{3}$
$b[3] * a[3, 2] * c[2] = \frac{1}{6}$

Es decir se cumplen las ecuaciones 1), 2) y 3) anteriores, por tanto el método en cuestión, denominado método de Kutta, es de tercer orden. Apliquémoslo, en primer lugar, al problema anterior con $N = 10$, $h = 1/N = 0{,}1$

```
(%i32) kill(all)$numer:true$
       N:10$ h:1/N$ y[0]:1$ t[0]:0$
       kill(f,k1,k2,k3)$
       define(f(t,y),y);
       define(k1(t,y),f(t,y));
       define(k2(t,y),f(t+0.5*h,y+0.5*h*k1(t,y)));
       define(k3(t,y),f(t+h,y+h*(-k1(t,y)+2*k2(t,y))));
       for k:1 thru N do (
       t[k]:t[0]+k*h,y[k]:y[k-1]+(h/6)*(k1(t[k-1],y[k-1])+
       +4*k2(t[k-1],y[k-1])+k3(t[k-1],y[k-1])))$
       print("Solución numérica: ",sol:makelist([t[j],y[j]],j,0,N))$
       wxplot2d([[discrete,sol],%e^t],[t,0,1],[style,[points,2,2],
       [lines,1,1]],[legend,"y(t) aproximada","y(t) exacta"],
       [xlabel,"t"],[ylabel,"y"]);
       print("El error global en 1.0 es EG(1.0) = ",%e-y[N])$
```

(%o7) $f(t, y) := y$
(%o8) $k1(t, y) := y$
(%o9) $k2(t, y) := 1{,}05y$
(%o10) $k3(t, y) := 1{,}11y$
Solución numérica: [[0,1],[0.1,1.105166666666667],[0.2,1.221393361111111],
[0.3,1.349843229587963],[0.4,1.491801742566297],[0.5,1.648689559159519],
[0.6,1.822076744464462],[0.7,2.013698482090641],[0.8,2.22547243912384],
[0.9,2.459517957305031],[1.0,2.71817726248161]]

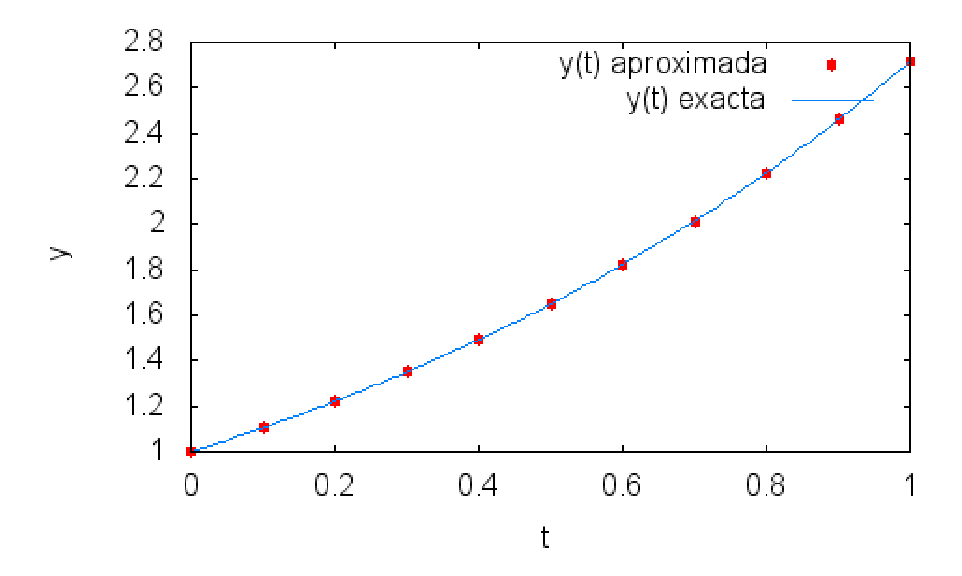


Figura 6.16: Exacta vs aproximada por RK(3) dado con h = 0.1

(%o13) El error global en 1,0 es $EG(1{,}0) = 1{,}0456597743546681 \cdot 10^{-4}$

Observación. Se observa que da un error global similar al método de Taylor del mismo orden, en este caso particular el mismo, también ocurre esto cuando tomamos N = 100, es decir dividimos h por 10. Asimismo, el error global funciona como era de esperar, se propone escribir este programa en un block y aplicarlo a ecuaciones diferentes, en las que los errores globales se comportan de un modo similar pero no tienen porque ser idénticos en uno u otro método. Un buen ejercicio puede ser aplicarlo a un PVI de segundo orden, escribiéndolo previamente como un sistema de dos ecuaciones de primer orden.

6.3.2. Método RK(4)“clásico” y aplicaciones

Los pasos a seguir son los descritos al comienzo del párrafo anterior. Dividiendo el programa en varios bloques, con el fin de que sea válido para otros casos con las modificaciones oportunas en el bloque que corresponda. Así, en el primer bloque, hemos de definir el PVI a integrar, es decir los valores iniciales, la longitud del intervalo de integración y la ecuación diferencial, en el segundo hemos de introducir el tablero de Butcher correspondiente al método Runge-Kutta explícito de cuatro etapas y orden cuatro “clasico” que vamos a utilizar, luego debemos programar el esquema del método y finalmente la salida deseada, que puede ser el valor aproximado de la solución en el

punto final del intervalo o los valores aproximados en toda la red de puntos considerada y/o su representación gráfica.

Así pues, comenzamos introduciendo los valores que definen el problema que vamos a integrar. Es necesario especificar la función $f(t,y)$ que define la ecuación o sistema diferencial ordinario $y' = f(t,y)$, así como el punto inicial (t_0, y_0). También incluiremos el tiempo final $TiempoFinal = t_0 + T$, de manera que T es la amplitud del intervalo de integración. Si queremos cambiar de problema tendremos que modificar estos valores. En primer lugar aplicaremos al problema anterior el RK "clasico" de 4 etapas y orden 4, cuyo tablero de Butcher está dado por

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array}$$

En cada paso el esquema es el mismo para cualquier método Runge-Kutta: se calculan los valores auxiliares ki y a partir de ellos se actualiza la solución y_n. También hay que actualizar el valor temporal $t_n = t_{n-1} + h$. Si es preciso, para evitar que la memoria se desborde se pueden sobreescribir tanto t_n como y_n, no lo hacemos ahora por si queremos representarla gráficamente. Si lo aplicamos al PVI anterior en [0,1],tomando N subintervalos o sea $h = 1/N$ y damos el EG en 1.0 el programa se puede escribir mediante el block que sigue:

```
(%i16) RK4_0(N,T):= block(numer:true,
       kill(t,y,f,k1,k2,k3,k4),h:T/N,t[0]:0.0,y[0]:1.0,
       define(f(t,y),y),
       define(k1(t,y),f(t,y)),
       define(k2(t,y),f(t+0.5*h,y+0.5*h*k1(t,y))),
       define(k3(t,y),f(t+0.5*h,y+0.5*h*k2(t,y))),
       define(k4(t,y),f(t+h,y+h*k3(t,y))),
       for n:1 thru N do (
       t[n]:t[0]+n*h,y[n]:y[n-1]+(h/6)*(k1(t[n-1],y[n-1])+
       2*(k2(t[n-1],y[n-1])+k3(t[n-1],y[n-1]))+k4(t[n-1],y[n-1]))),
       sol:makelist([t[j],y[j]],j,0,N),
       wxplot2d([[discrete,sol],%e^t],[t,0,1],[style,[points,2,2],
       [lines,1,1]],[legend,"y(t) aproximada","y(t) exacta"],
       [xlabel,"t"],[ylabel,"y"]), display(y[N]),
       print("El error global en 1.0 es EG(1.0) = ",%e-y[N])
       )$
```

Ahora lo ejecutamos con $N = 10$ y $N = 100$, o sea con $h = 0{,}1$ y $h = 0{,}01$ y vemos como el error global se comporta como era de esperar, es decir se divide aproximadamente por 10^4.

```
(%i17) RK4_0(10,1)$
```

(%t17)

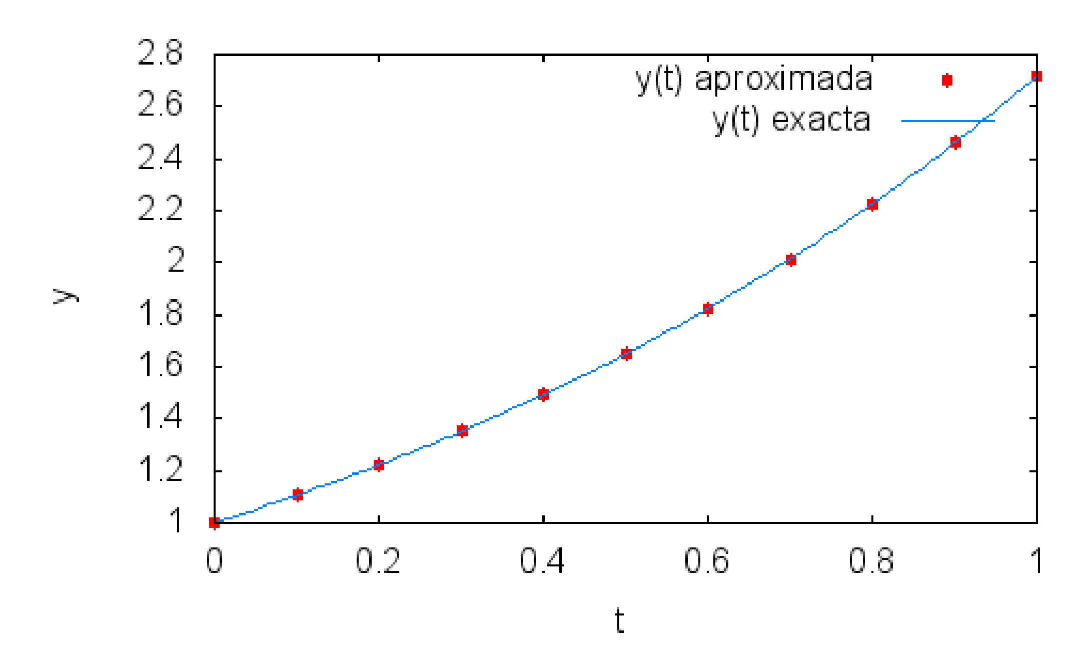


Figura 6.17: Exacta vs aproximada por RK(4) "clásico" con h = 0.1

$y_{10} = 2{,}718279744135166$ es el valor aproximado para $y(1{,}0)$ dado por el RK(4) "clásico" con 10 pasos de tamaño 0,1. En tanto que el error global en 1.0 dado por $EG(1{,}0) = y(1{,}0) - y_{10}$ es $EG(1{,}0) = 2{,}0843238792700447 \cdot 10^{-6}$, que muestra la bondad del método, pues con pocos cálculos se ha obtenido una buena aproximación. Seguidamente ejecutamos el mismo programa con paso $h = 0{,}01$.

```
(%i18) Esquema0RK4(100,1)$
```

(%t18)

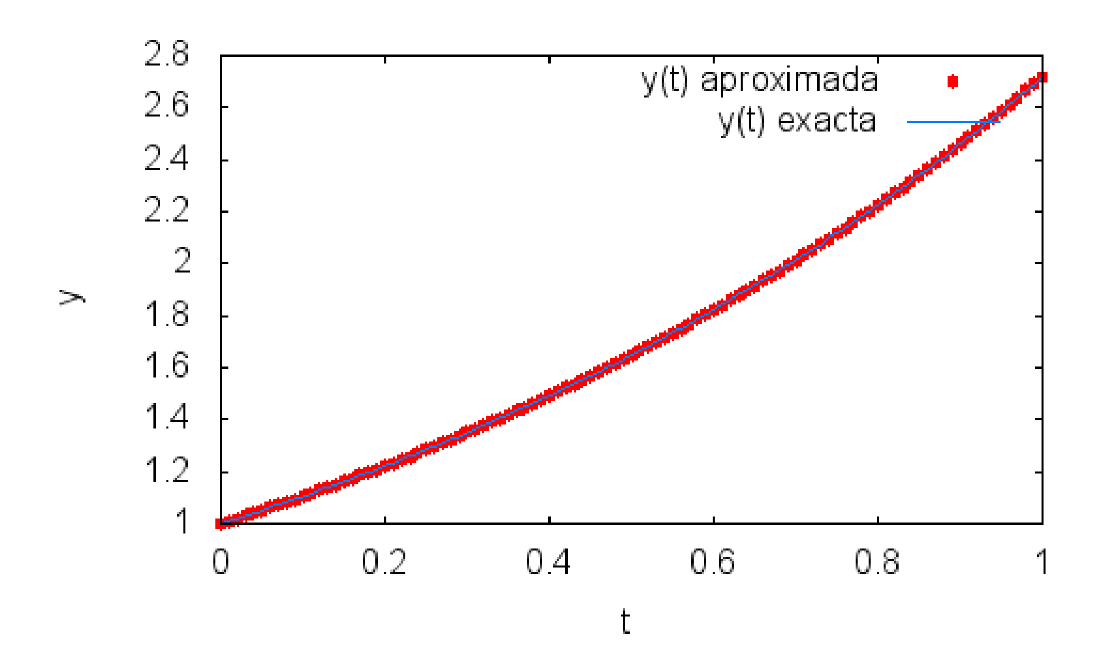


Figura 6.18: Exacta vs aproximada por RK(4) "clásico" con h = 0.01

$y_{100} = 2{,}718281828234404$ es el valor aproximado para $y(1{,}0)$ dado por el RK(4) "clásico" con 100 pasos de tamaño 0,01. Siendo ahora el error global en 1.0 es $EG(1{,}0) = 2{,}2464119453502462 \cdot 10^{-10}$.

6.3.3. El paquete diffeq y el comando rk

Seguidamente cargaremos el paquete "**diffeq**", que nos permitirá usar el comando "**rk**" para integrar PVI para edos, ya sea una ecuación o un sistema de ecuaciones de primer orden. Veamos información sobre el mismo a continuación mediante el comando "**? rk**" y su aplicación al problema que nos ocupa, este utiliza el método RK clásico de cuatro etapas y orden cuatro.

```
(%i19) load(diffeq);
```

(%o19) $C : /PROGRA\ 2/MAXIMA\ 1{,}0-2/share/maxima/5{,}28{,}0-2/share/numeric/diffeq.mac$

```
(%i20) ? rk;
```

– Función: rk (EDO, var, inicial, dominio)

– Función: rk ([EDO1,...,EDOm], [v1,...,vm], [inic1,...,inicm], dominio)
La primera forma se usa para resolver numéricamente una ecuación diferencial ordinaria de primer orden (EDO), y la segunda forma resuelve numéricamente un sistema de $< m >$ de esas ecuaciones, usando el método de Runge-Kutta de cuarto orden. $< var >$ representa la variable dependiente. EDO debe ser una expresión que dependa únicamente de las variables independiente y dependiente, y define la derivada de la variable dependiente en función de la variable independiente.
La variable independiente se representa con $< dominio >$, que debe ser una lista con cuatro elementos, como por ejemplo: $[t, 0, 10, 0{,}1]$, **el primer elemento de la lista identifica la variable independiente, el segundo y tercer elementos son los valores inicial y final para esa variable, y el último elemento da el valor de los incrementos** que deberán ser usados dentro de ese intervalo.
Si se van a resolver $< m >$ ecuaciones, deberá haber $< m >$ variables dependientes $v1$, $v2$, ..., vm . Los valores iniciales para esas variables serán $< inic1 >$, $< inic2 >$, ..., $< inicm >$. Continuará existiendo apenas una variable independiente definida por la lista $< domain >$, como en el caso anterior. $< EDO1 >, ..., < EDOm >$ son las expresiones que definen las derivadas de cada una de las variables dependientes en función de la variable independiente. Las únicas variables que pueden aparecer en cada una de esas expresiones son la variable independiente y cualquiera de las variables dependientes. Es importante que las derivadas $< EDO1 >, ..., < EDOm >$ sean colocadas en la lista en el mismo orden en que fueron agrupadas las variables dependientes; por ejemplo, el tercer elemento de la lista será interpretado como la derivada de la tercera variable dependiente.
El programa intenta integrar las ecuaciones desde el valor inicial de la variable independiente, hasta el valor final, usando incrementos fijos. Si en algún paso una de las variables dependientes toma un valor absoluto muy grande, la integración será suspendida en ese punto. El resultado será una lista con un número de elementos igual al número de iteraciones realizadas. Cada elemento en la lista de resultados es también una lista con $< m > +1$ elementos: el valor de la variable independiente, seguido de los valores de las variables dependientes correspondientes a ese punto.
There are also some inexact matches for ‘rk’. Try ‘?? rk’ to see them.

(%o20) *true*

Por tanto, lo primero de todo es cargar este paquete mediante “**load(diffeq)**” y utilizar el comando de Maxima “**rk(f, y, y0, [t, t0, t1, h])**” para ecuaciones escalares o sistemas, veamos su aplicación al problema text que nos ocupa con $h = 0{,}1$ y $h = 0{,}01$.

```
(%i21) kill(all)$
       load(diffeq)$
       kill(f)$
       f(t,y):=y;
       print("Solución numérica: ")$
       numsol:rk(f(t,y),y,1,[t,0,1,0.1]);
       print("Representación gráfica de la solución numérica: ")$
       wxplot2d([discrete,numsol],[style,points])$
```

(%o3) $f(t, y) := y$
Solución numérica:
(%o5)
[[0, 1], [0,1, 1,105170833333333], [0,2, 1,221402570850695],
[0,3, 1,349858497062538], [0,4, 1,491824240080686], [0,5, 1,648720638596838],
[0,6, 1,822117962091933], [0,7, 2,013751626596777], [0,8, 2,225539563292315],
[0,9, 2,459601413780071], [1,0, 2,718279744135166]]
Representación gráfica de la solución numérica:
(%t7)

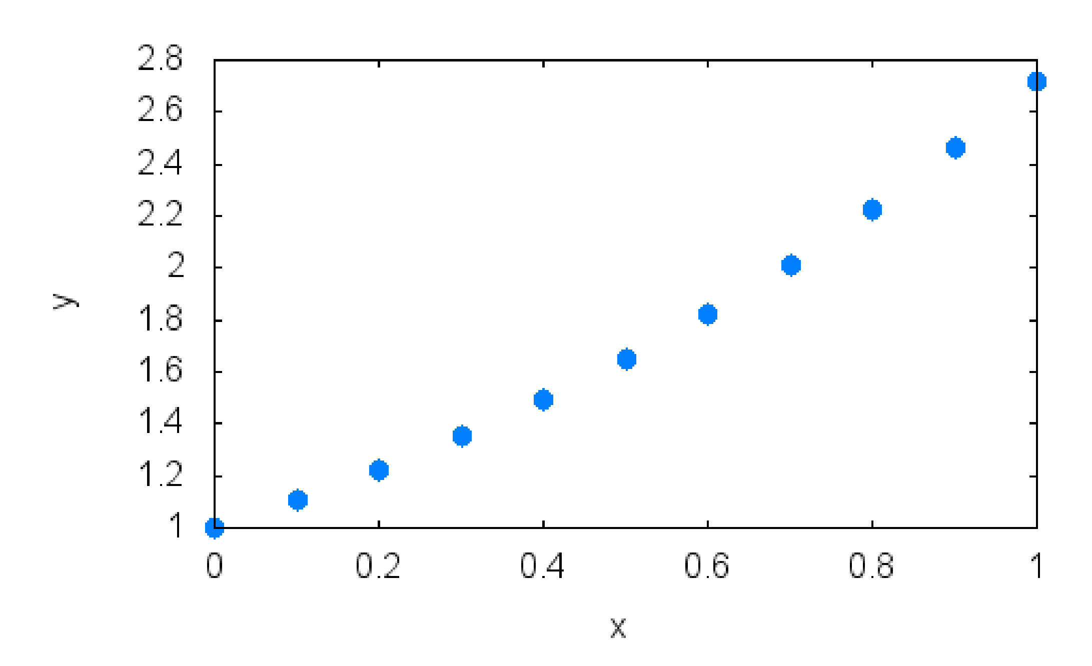


Figura 6.19: Solución aproximada por RK(4) "clásico" con h = 0.1

En tanto que el error global en 1,0 resulta ser

```
(%i8)  print("El error global en 1.0 es EG(1.0) = ",
       float(%e-2.718279744135166))$
```

El error global en 1,0 es $EG(1{,}0) = 2{,}08432387927004471\dot{0}^{-6}$, el mismo que antes obtuvimos, pues el comando "rk" lleva implementado el mismo método de integración numérica. Repitiendo ahora con paso $h = 0{,}01$, tendremos:

```
(%i9)  kill(f)$
       f(t,y):= y;
       solnum:rk(f(t,y),y,1.0,[t,0.0,1.0,0.01]);
       wxplot2d([discrete,solnum],[style,points])$
```

(%o10) $\mathrm{f}\,(t, y) := y$

(%o11) Omitimos la salida numérica para abreviar.

(%t12)

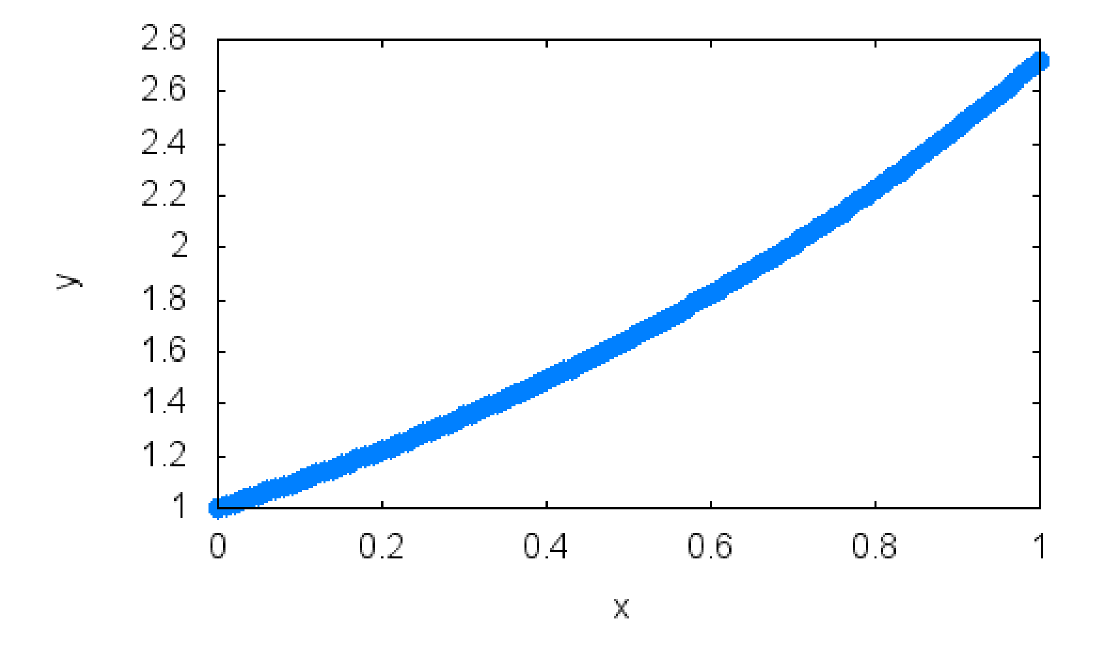


Figura 6.20: Solución aproximada por RK(4) "clásico" con h = 0.01

Y para el error global se tiene

```
(%i13) print("El error global en 1.0 es EG(1.0) = ",
       float(%e-2.718281828234403))$
```

El error global en 1,0 es $EG(1{,}0) = 2{,}2464208271344432 \cdot 10^{-10}$, que coincide con el obtenido antes por la razón señalada.

6.3.4. Aplicación del RK(4) "clásico" al problema de dos cuerpos plano

En este último apartado, en vez de trabajar con el problema que venimos estudiando, vamos a considerar el problema plano de dos cuerpos puntuales, que se mueven atraídos según la ley de la gravitación universal de Newton. En unidades apropiadas y con la ayuda del editor de Maxima, las ecuaciones del movimiento adoptan la forma:

```
(%i14) numer:false$ kill(all)$
       'diff(y1,t) = y3;
       'diff(y2,t) = y4;
       'diff(y3,t) = -y1/(y1^2+y2^2)^(3/2);
       'diff(y4,t) = -y2/(y1^2+y2^2)^(3/2);
```

(%o1) $\frac{d}{dt}y1 = y3$

(%o2) $\frac{d}{dt}y2 = y4$

(%o3) $\frac{d}{dt}y3 = -\frac{y1}{\left(y2^2+y1^2\right)^{3/2}}$

(%o4) $\frac{d}{dt}y4 = -\frac{y2}{\left(y2^2+y1^2\right)^{3/2}}$

Siendo $(y1, y2)$ las coordenadas de uno de los cuerpos (satélite) en un sistema con origen en el otro cuerpo central (planeta), en tanto que $(y3, y4)$ representa el vector velocidad del cuerpo satélite.

Como recordaréis por la primera ley de Kepler: "los planetas describen órbitas elípticas alrededor del sol, que ocupa uno de los focos de esa elipse".

En general, las soluciones del problema de dos cuerpos son cónicas (pueden ser elipses, parábolas o hipérbolas), ahora bien si se toman las condiciones iniciales siguientes: $y1(0) = 1 - e$, $y2(0) = y3(0) = 0$, e $y4(0) = \sqrt{\frac{1+e}{1-e}}$ (con $0 \leq e < 1$), se puede probar que la solución es una elipse de excentricidad e, como es el caso en el sistema planetario solar. Normalmente, no aparecen parámetros en las condiciones iniciales, pero esto permite abordar un conjunto de problemas con diferencias cualitativas significativas. En particular, si $e = 0$ el satélite describe una órbita circular, entorno al cuerpo central,

con periodo 2π, así pues si suponemos que en el instante inicial el satélite se encuentra en el punto $(1{,}0, 0{,}0)$ con velocidad $v = (0{,}0, 1{,}0)$. El esquema para integrar numéricamente este problema con el comando "rk" es similar al de antes, pero ahora es una función vectorial de cuatro componentes, puesto que la solución es periódica con periodo 2π, si integramos de 0 a 2π, tras una vuelta completa, debemos tener valores próximos a los iniciales, veamos esto en el programa que sigue, en el que realizamos 20 pasos de amplitud $h = \frac{2\pi}{20} = \frac{\pi}{10} = 0{,}31415926535898$ y mostramos sólo el último valor que calcula.

```
(%i8) /* Con 20 pasos */
      kill(all)$ load(diffeq);numer:true$
      y:[y1,y2,y3,y4];
      f(t,y):=[y3,y4,-y1/(y1^2+y2^2)^(3/2),
      -y2/(y1^2+y2^2)^(3/2)];
      h=%pi/10;
      discretesolution1:rk(f(t,y),y,[1.0,0.0,0.0,1.0],
      [t,0,2*%pi,%pi/10])$ last(discretesolution1);
```

(%o1) $C:/PROGRA\ 2/MAXIMA\ 1{,}0-2/share/maxima/5{,}28{,}0-2$ $/share/numeric/diffeq.mac$

(%o3) $[y1, y2, y3, y4]$

(%o4) $\mathrm{f}\left(t, y\right) := [y3, y4, \frac{-y1}{\left(y1^2+y2^2\right)^{\frac{3}{2}}}, \frac{-y2}{\left(y1^2+y2^2\right)^{\frac{3}{2}}}]$

(%o5) $h = 0{,}31415926535898$

(%o7) $[6{,}283185307179586, 0{,}99944519714206, 0{,}0038657399331735,$ $-\,0{,}0038695061807764, 1{,}000266720696481]$

Observación.- La salida del programa anterior muestra, solamente, el último elemento calculado de la solución numérica (que hemos denominado discretesolution1) proporcionada por el comando "rk", mediante la instrucción last(discretesolution1), que nos da [6.283185307179586,0.99944519714206, 0.0038657399331735,-0.0038695061807764,1.000266720696481], cuyos cuatro últimos elementos son los valores aproximados de $y1, y2, y3, y4$ para $t = 20h = 6{,}283185307179586 \leq 2\pi$, que lo hace puesto que en los redondeos resulta $20h \leq 2\pi$ (si al ir añadiendo h, debido a los redondeos, se pasará del valor final de la variable contemplada como tal en dominio no calcularía dichos valores en tal caso y tendríamos que hacer un ajuste para obtener esta aproximación como veremos en el ejemplo que sigue). Ahora, el error global de esta aproximación en 2π será:

```
(%i8) print("El error global con 20 pasos en 2*%pi es
      EG(",2*%pi,") = ", [1.0,0.0,0.0,1.0]- [0.99944519714206,
      0.0038657399331735,-0.0038695061807764,1.000266720696481])$
```

El error global con 20 pasos en $t = 2\pi$ es $EG(6{,}283185307179586) = [5{,}548028579399622 \cdot 10^{-4}, -0{,}0038657399331735, 0{,}0038695061807764, -2{,}6672069648103758 \cdot 10^{-4}]$.

Repitamos ahora la integración con paso $h = \frac{2\pi}{200} = \frac{\pi}{100} = 0{,}031415926535898$, resultará:

```
(%i9) /*Con 200 pasos */ kill(all)$ load(diffeq);
      numer:true$ kill(f)$
      y:[y1,y2,y3,y4];
      f(t,y):=[y3,y4,-y1/(y1^2+y2^2)^(3/2),-y2/(y1^2+y2^2)^(3/2)];
      h=%pi/100;
      discretesolution2:rk(f(t,y),y,[1.0,0.0,0.0,1.0],
      [t,0,2*%pi,%pi/100])$ last(discretesolution2);
```

(%o1) $C : /PROGRA\ 2/MAXIMA\ 1{,}0-2/share/maxima/5{,}28{,}0-2/share/numeric/diffeq.mac$

(%o4) $[y1, y2, y3, y4]$

(%o5) $\mathrm{f}(t, y) := [y3, y4, \dfrac{-y1}{(y1^2+y2^2)^{\frac{3}{2}}}, \dfrac{-y2}{(y1^2+y2^2)^{\frac{3}{2}}}]$

(%o6) $h = 0{,}031415926535898$

(%o8) [6.251769380643689,0.9995065602185,-0.031410593663266, 0.03141059439279,0.99950656822774]

Observación. Podemos apreciar que la última salida de (%o8) es igual a [6.251769380643689,0.9995065602185,-0.031410593663266,0.03141059439279, 0.99950656822774] que da un error similar o peor que la anterior, a pesar de haber dividido el paso por 10. De hecho, el tiempo final para el que calcula la aproximación es 6.251769380643689 distinto de 2π, y esto es debido a que por los errores de redondeo ahora se tiene que $6{,}251769380643689 + h > 2\pi$, por lo que no calcula el valor aproximado de la solución para $t = 2\pi$. Como podemos comprobar a continuación:

```
(%i9) is(6.251769380643689+0.031415926535898>2*%pi);
```

(%o9) *true*

```
(%i10) 2*%pi;
```

(%o10) 6,283185307179586

Es pues necesario un ajuste del último paso, para obtener una mejor aproximación de la solución en 2π, haciendo un sólo paso de amplitud $2\pi - 6{,}251769380643689$, partiendo del valor al que hemos llegado, lo que haríamos en la forma siguiente:

```
(%i11) kill(all)$ load(diffeq);
       numer:true$ kill(f)$
       y:[y1,y2,y3,y4];
       f(t,y):=[y3,y4,-y1/(y1^2+y2^2)^(3/2),-y2/(y1^2+y2^2)^(3/2)];
       ajustesolution:rk(f(t,y),y,[0.9995065602185,-0.031410593663266,
       0.03141059439279,0.99950656822774],
       [t,6.251769380643689,2*%pi,2*%pi-6.251769380643689]);
```

(%o1) C:/PROGRA 2/MAXIMA 1.0-2/share/maxima/ 5.28.0-2/share/numeric/diffeq.mac

(%o4) $[y1, y2, y3, y4]$

(%o5) $\mathrm{f}\,(t, y) := [y3, y4, \dfrac{-y1}{\left(y1^2 + y2^2\right)^{\frac{3}{2}}}, \dfrac{-y2}{\left(y1^2 + y2^2\right)^{\frac{3}{2}}}]$

(%o6) $[[6{,}251769380643689, 0{,}9995065602185, -0{,}031410593663266, 0{,}03141059439279, 0{,}99950656822774], [6{,}283185307179586, 0{,}99999999465787, 1{,}6532531159352271 \cdot 10^{-7}, -1{,}6532530891510966 \cdot 10^{-7}, 1{,}000000002671051]]$

Obteniendo ahora el error global siguiente

```
(%i7) print("El error global en 2*%pi es EG(",2*%pi,") = ",
      float([1.0,0.0,0.0,1.0]- [0.99999999465787,
      1.6532531159352271*10^-7,-1.6532530891510966*10^-7,
      1.000000002671051]))$
```

El error global en 2π es $EG(2\pi) = [5{,}3421299606171146 \cdot 10^{-9}, -1{,}6532531159352271 \cdot 10^{-7}, 1{,}6532530891510966 \cdot 10^{-7}, -2{,}6710509359872958 \cdot 10^{-9}]$.

6.4. Ejercicios propuestos

Completar con Maxima los ejercicios siguientes:

1. Hallar, si existe, algún método Runge-Kutta explícito de tres etapas y orden máximo, cuyo tablero de Butcher tenga la forma

$$\begin{array}{c|ccc} 0 & & & \\ 1/3 & 1/3 & & \\ c_3 & a_{31} & a_{32} & \\ \hline & 1/4 & b_2 & 3/4 \end{array}$$

   Aproximar la solución en el tiempo $t = 1{,}0$ del PVI: $y'(t) = t.y(t)$, $y(0) = 1$; tomando $h = 0{,}1$ e integrándolo mediante el método RK(3) hallado en el apartado anterior. Asimismo, tras integrar por dicho método con paso de amplitud $h = 0{,}2$, estimar, por el método de extrapolación de Richardson, el error cometido en la primera aproximación hallada de $y(1{,}0)$ y mejorar esta mediante dicho método de extrapolación. Comparar la solución numérica obtenida con paso $h = 0{,}1$ con la exacta dada por $y(t) = e^{t^2/2}$ en el punto $t = 1{,}0$, realizando para ello las gráficas de las soluciones discreta y continua.

2. Para el PVI del ejercicio anterior, repite lo pedido en el mismo pero utilizando el método de Taylor de orden tres y el de orden cuatro.

3. Finalmente, repite el anterior utilizando el método Runge-Kutta explícito de cuatro etapas y orden 4 “clásico”, con un programa realizado por vosotros y con el comando rk.

Bibliografía

[1] Jerónimo Alaminos Prats, Camilo Aparicio del Prado, José Extremera Lizana, Pilar Muñoz Rivas, Armando R. Villena Muñoz: Prácticas de ordenador con wxMaxima. Universidad de Granada, 2010.

[2] A. Jesús Arriaza Gómez, José María Calero Posada, Loreto Del Águila Garrido, Aurora Fernández Valles, Fernando Rambla Barreno, María Victoria Redondo Neble y José Rafael Rodríguez Galván: Pácticas de Matemáticas con Maxima (Matemáticas usando software libre). Universidad de Cadiz, curso 2008-2009.

[3] David López Medina: Curso/taller de Maxima y Prácticas de Cálculo Numérico. Departamento de Matemática Aplicada y Estadística, Universidad Politécnica de Cartagena, 2012.

[4] Equipo de desarrollo de Maxima: Manual de Maxima (versión en español).

[5] José Manuel Mira: Elementos para prácticas con Maxima. Departamento de Matemáticas, Universidad de Murcia, http://webs.um.es/mira

[6] Escuela Politécnica de Ingeniería: Prácticas de Cálculo con Maxima. Universidad de Oviedo, 2010.

[7] J. Rafael Rodríguez Galván: Maxima con wxMaxima: sofware libre en el aula de Matemáticas. Departamento de Matemáticas de la Universidad de Cádiz, 2007.

[8] Mario Rodríguez Riotorto: Primeros pasos en maxima. http://riotorto.users.sourceforge.net, 2011.

[9] G. Soler: Prácticas de Fundamentos Matemáticos. Departamento de Matemática Aplicada y Estadística, Universidad Politécnica de Cartagena, 2010.

[10] A. Vigueras: Cálculo Numérico (Teoría, Problemas y algunos programas con Maxima), Universidad Politécnica de Cartagena, 2016.